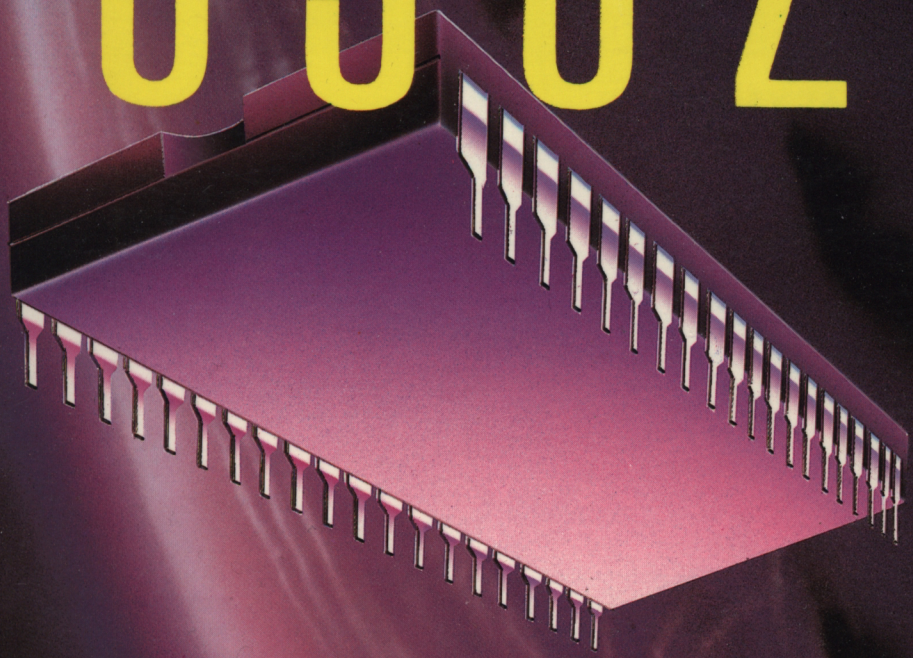
RODNAY ZAKS

PROGRAMMATION DU 6502



SYBEX

PROGRAMMATION DU 6502

RODNAY ZAKS

Paris • Berkeley • Düsseldorf

Traduction française : Daniel-Jean David

Couverture : Nicolae Razumieff.

Tous les efforts ont été faits pour fournir une information complète et actualisée. Cependant, Sybex n'assume aucune responsabilité pour son utilisation ou pour toute contrefaçon de brevets ou autres droits de tierces parties qui pourraient en résulter. Les fabricants ne donnent aucune autorisation sur les brevets ou droits et se réservent le droit de changer les circuits à tout moment sans avis préalable. En particulier les caractéristiques techniques et les prix sont sujets à changement rapide. Les comparaisons et évaluations sont données à titre indicatif et comme principes directeurs. Le lecteur est prié de se référer aux données du fabricant pour les spécifications exactes.

Copyright version originale © 1978, 1979, 1980, Sybex Inc.
version française © 1979, 1980, Sybex

3ème édition

Toute reproduction même partielle, par quelque procédé que ce soit, interdite sans autorisation écrite préalable. Une copie par xérographie, photographie, film, bande magnétique ou autre constitue une contrefaçon passible des peines prévues par la loi sur la protection des droits d'auteur.

Imprimé en France. World rights reserved.

ISBN : 2-902414-24-2, 2e édition
ISBN : 2-902414-13-7, 1re édition
ISBN : 2-7361-0031-3, 3e édition

TABLE DES MATIERES

PREFACE 5

1. CONCEPTS DE BASE 7

Ce qu'est la programmation, l'organigramme, l'algorithme, la représentation des programmes, des données.

2. ORGANISATION MATERIELLE DU 6502 35

L'architecture du système, l'organisation interne du 6502 — le cycle instruction-exécution. Recherche et instruction suivante. Les registres du 6502.

3. TECHNIQUES DE PROGRAMMATION 49

Programmes arithmétiques, addition, soustraction, BCD arithmétique, multiplication, division, subroutines.

4. JEU D'INSTRUCTION DU 6502 93

Classes d'instruction : transfert de données, traitement de données, test, entrées-sorties, contrôle, instructions disponibles sur le 6502.

5. TECHNIQUES D'ADRESSAGE 181

Adressage : implicite, immédiat, absolu, relatif, indexé (pré- et post-), indirect. Modes combinés. Programmes type.

6. TECHNIQUES D'ENTREES-SORTIES 203

Entrées-sorties de base. Production d'un signal de retard. Impulsion. Tests. Conversion parallèle-série : communication avec des systèmes E-S. « Hand shaking », LED, télétype, sondage, interruptions, etc.

7. COMPOSANTS D'ENTREES-SORTIES 243

Le PIO standard, le 6520, le 6522, le 6532.

8. EXEMPLES D'APPLICATIONS 251

Effacement de mémoire, lecture de caractères, test d'un caractère, une parenthèse, générateur de bit de parité, conversion de code, élément le plus grand, la somme d'une table, « checksum », compter les zéros.

9. STRUCTURES DE DONNEES 263

1ère Partie — Concepts

Pointeurs, listes, fichier séquentiel, file d'attente, registres, recherche et mise en mémoire.

2ème Partie — Exemples

Table, table simple, recherche, insertion, suppression, liste alphabétique, recherche, insertion, suppression, liste chaînée, recherche, insertion, suppression, arbre binaire, hachage, tri à bulle, fusion.

10. DEVELOPPEMENT DES PROGRAMMES 327

Choix de programmes, aides au développement de software, alternative hardware, les ressources matérielles, l'assembleur, les macros et autres facilités.

11. CONCLUSION 353

APPENDICES 356

A. Table de conversion en hexadécimal 356
B. Jeu d'instructions du 6502 : liste alphabétique 357
C. Jeu d'instructions du 6502 : hexadécimal et durée ... 358
D. Jeu d'instructions du 6502 : valeurs binaires 360
E. Table ASCII 361
F. Table de branchements relatifs 362
G. Liste par code opération hexadécimal 363
INDEX 365
BIBLIOGRAPHIE SYBEX

PREFACE

Ce livre a été conçu pour former un texte complet et autonome en vue de l'apprentissage de la programmation à l'aide du 6502. Il peut être lu par un utilisateur qui n'aurait jamais programmé avant, et il devrait être utile aussi à toute personne qui utilise le 6502.

Pour le lecteur qui a déjà programmé, ce livre enseignera les techniques de programmation qui dépendent ou tirent parti des caractéristiques spécifiques du 6502. Le texte couvre les techniques élémentaires ou intermédiaires nécessaires pour commencer à programmer de façon effective.

Ce texte a pour but de mener à un bon niveau de compétence le lecteur qui souhaite programmer ce microprocesseur. Naturellement, aucun livre ne peut enseigner la programmation effective s'il n'est pas accompagné de pratique. Toutefois, ce livre devrait mener le lecteur au point où il se sente capable de programmer par lui-même et résoudre des problèmes simples ou modérément complexes à l'aide d'un microordinateur.

Ce livre est fondé sur l'expérience de l'auteur qui a enseigné la programmation des microordinateurs à plus de 1000 personnes. Il en résulte qu'il est très structuré. Les chapitres vont généralement du simple au complexe. Les lecteurs qui ont déjà appris la programmation élémentaire peuvent passer le chapitre d'introduction. Pour ceux qui n'ont jamais programmé, les dernières sections de certains chapitres peuvent nécessiter une seconde lecture. Le livre a été conçu pour mener le lecteur systématiquement à travers tous les concepts fondamentaux et les techniques requises pour construire des programmes de plus en plus complexes. Il est donc conseillé de bien suivre l'ordre des chapitres. En outre, pour obtenir des résultats effectifs, il est important que le lecteur essaie de résoudre le plus grand nombre d'exercices possibles. Leur difficulté a été soigneusement graduée. Ils ont pour but de vérifier que les notions présentées ont bien été comprises. Si l'on ne fait pas les exercices de programmation, il est impossible de profiter de toute la valeur éducative de l'ouvrage. Certains des exercices peuvent prendre du temps, comme, par exemple, celui sur la multiplication. Toutefois, en faisant les exercices, vous programmerez véritablement et ferez un « *apprentissage par l'action* ». C'est indispensable.

Pour ceux qui auront pris goût à la programmation en arrivant à la fin de ce volume, un ouvrage complémentaire est disponible : « Le livre des applications du 6502 », qui couvre de manière détaillée les entrées-sorties et l'utilisation effective d'une carte utilisant le 6502.

Enfin, le livre « jeux sur le 6502 » présente des programmes complexes utilisant le 6502 pour la réalisation de divers jeux.

D'autres livres de cette série traitent de la programmation pour d'autres microprocesseurs.

Enfin, ceux qui souhaitent compléter leurs connaissances en hardware, peuvent consulter les livres de référence « Microprocesseurs » (Réf. C4) et « Techniques d'interface » (Réf. C5).

L'auteur recevra avec reconnaissance tout commentaire émanant des lecteurs. Toutes suggestions d'améliorations telles que programmes supplémentaires souhaités, développés ou jugés intéressants par les lecteurs, seront appréciées.

1

CONCEPTS DE BASE

INTRODUCTION

Ce chapitre introduit les concepts et les définitions de base concernant la programmation des ordinateurs. Le lecteur déjà familier de ces notions souhaitera peut-être passer rapidement sur ce chapitre et aller tout de suite au chapitre II. Cependant, on suggère même au lecteur expérimenté de lire ce chapitre introductif : on y présente de nombreuses notions importantes, comprenant, par exemple, le complément à deux, le DCB et d'autres représentations.

Certains de ces concepts seront probablement nouveaux pour le lecteur ; d'autres peuvent s'ajouter aux connaissances de programmeurs expérimentés.

QU'EST-CE QUE LA PROGRAMMATION ?

Etant donné un problème, on doit d'abord trouver une solution. L'expression de cette solution sous la forme d'une procédure à suivre pas à pas s'appelle un *algorithme.* Un algorithme est la spécification pas à pas de la solution d'un problème donné. Il doit en outre se terminer au bout d'un nombre fini d'étapes. Il peut être exprimé dans n'importe quel langage.

Un algorithme typique est par exemple celui-ci :

1 — Insérer la clé dans le trou de la serrure
2 — Tourner la clé d'un tour complet à gauche
3 — Prendre le bouton de porte
4 — Tourner le bouton à gauche et
5 — Pousser la porte.

A ce moment, si l'algorithme est correct pour le type de serrure considéré, la porte va s'ouvrir. Cette procédure en 5 étapes constitue un algorithme d'ouverture de la porte.

Une fois que la solution d'un problème a été exprimée sous la forme d'un algorithme, celui-ci doit être exécuté par l'ordinateur.

Malheureusement, c'est maintenant un fait bien connu que les ordinateurs ne peuvent comprendre, ni exécuter un programme exprimé en français (ou dans tout autre langage humain). Cela tient à l'*ambiguïté de la syntaxe* de toutes les langues humaines naturelles. Seul, un sous-ensemble bien défini de la langue naturelle peut être « compris » par l'ordinateur. Cela constitue un *langage de programmation.*

On appelle programmation la conversion d'un algorithme en une séquence d'instructions appartenant à un langage de programmation. Pour être plus précis, la phase de traduction proprement dite de l'algorithme dans le langage de programmation s'appelle *codage.* La programmation désigne en fait non seulement le codage mais aussi la conception d'ensemble des programmes et des structures de données qui vont réaliser l'algorithme.

La programmation effective requiert non seulement la compréhension des techniques possibles de mise en œuvre des algorithmes standards mais aussi l'exploitation intelligente de toutes les ressources hardware de l'ordinateur, telles que les registres internes, les mémoires et les périphériques plus une certaine créativité dans l'utilisation des structures de données appropriées. Ces techniques seront traitées dans les chapitres suivants.

La programmation exige aussi une discipline stricte du côté de la documentation afin d'assurer que les programmes soient compréhensibles pour les autres autant que pour leur auteur. La documentation doit être à la fois extérieure et intérieure au programme.

La documentation interne au programme consiste en commentaires placés dans le corps du programme et qui expliquent son fonctionnement.

La documentation externe désigne les documents de conception qui sont séparés du programme : explications écrites, manuels et ordinogrammes.

ORDINOGRAMMES

Une étape intermédiaire est presque toujours marquée dans le passage de *l'algorithme* au *programme.* C'est l'*ordinogramme.*

Un ordinogramme n'est autre qu'une représentation symbolique de l'algorithme exprimée sous forme d'une suite de rectangles et de

losanges contenant les étapes de l'algorithme. Les rectangles sont utilisés pour des « ordres » ou « instructions exécutables ».

Les losanges sont utilisés pour des « tests » tels que « si l'information X est vraie, alors faire l'action A, sinon faire l'action B ». Plutôt que de présenter ici une définition formelle des ordinogrammes, nous introduirons et discuterons les ordinogrammes au moment où nous présenterons des programmes dans le cours du livre.

Le tracé de l'ordinogramme est une étape hautement conseillée entre la spécification de l'algorithme et le codage proprement dit de la solution. Il est remarquable que l'on ait observé que peut-être seulement 10 % de la population des programmeurs sont capables d'écrire un programme correct sans passer par l'ordinogramme. Malheureusement, on a observé aussi que 90 % de la population croient faire partie des 10 % qui n'ont pas besoin d'ordinogramme.

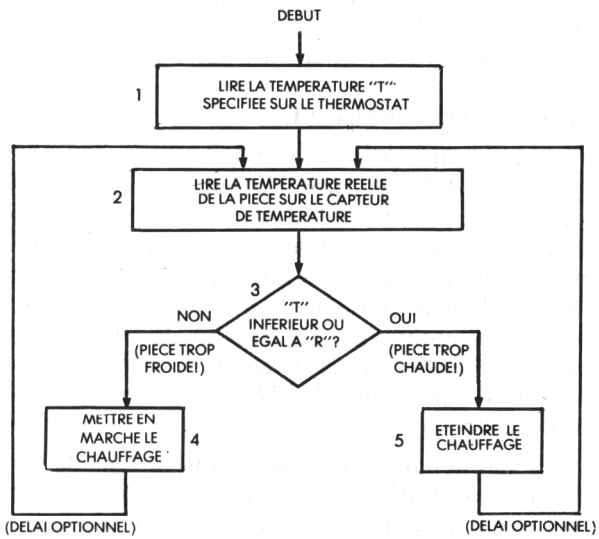


Fig. 1-1 : Ordinogramme de régulation de température

Il en résulte que en moyenne, 80 % des programmes ne fonctionneront pas à leur premier passage en machine. (Ces nombres n'ont bien sûr pas la prétention d'être très précis). En bref, la plupart des programmeurs novices voient rarement la nécessité de tracer un ordinogramme. Ceci entraîne habituellement des programmes « boîteux », ou erronés. Ils doivent alors passer un temps considéra-

ble à tester et à corriger leur programme (cela s'appelle la phase de *mise au point).* Il est donc fortement recommandé d'avoir la discipline de tracer l'ordinogramme dans tous les cas. Cela demandera un petit supplément de temps avant le codage, mais il en résultera habituellement un programme clair qui s'exécutera vite correctement. Lorsque les ordinogrammes sont bien compris, un faible pourcentage des programmeurs sont capables de franchir cette étape mentalement sans avoir à faire un tracé sur papier. Malheureusement, dans ce cas, les programmes qu'ils écriront seront difficiles à comprendre pour quelqu'un d'autre privé de la documentation que constitue l'ordinogramme. Il est donc universellement recommandé d'avoir la stricte discipline de tracer l'ordinogramme pour tout programme important. On en fournira de nombreux exemples dans tout le livre.

REPRESENTATION DE L'INFORMATION

Tous les ordinateurs manipulent des informations, sous forme de nombres ou de caractères. Nous examinons ici les représentations internes et externes des informations dans un ordinateur.

REPRESENTATION INTERNE

Dans un ordinateur, toute information est stockée sous forme de groupes de bits. *BIT* est l'abréviation de binary digit (chiffre binaire), c'est-à-dire « 0 » ou « 1 ». A cause des limitations de l'électronique conventionnelle, la seule manière pratique de représenter des informations fait appel à une logique à deux états représentant le « 0 » et le « 1 ». Il en résulte qu'aujourd'hui tout traitement d'information est effectué en binaire. Dans le cas des microprocesseurs en général, et en particulier du 6502, ces bits sont groupés par 8. Un groupe de 8 bits s'appelle un « *OCTET* ». Un groupe de quatre bits s'appelle un quartet.

Examinons maintenant la représentation interne de l'information binaire. Il y a deux entités à représenter à l'intérieur de la machine. La première est le programme, c'est-à-dire une séquence d'instructions.

La seconde est les données sur lesquelles le programme travaille et qui peuvent être des nombres ou du texte alphanumérique. Nous allons discuter ci-dessous trois représentations : programme, nombres, caractères alphanumériques.

1 — Représentation du programme

Toutes les instructions sont représentées à l'intérieur de la machine par un ou plusieurs octets. Les instructions dites « courtes » sont représentées par un seul octet. Les instructions « longues »

sont contenues dans deux octets ou plus. Comme le 6502 est un microprocesseur 8 bits, il cherche les octets un par un en mémoire. Par suite, les instructions en un octet ont toujours une probabilité de s'exécuter plus vite que les instructions à deux ou trois octets. On verra plus loin que c'est une caractéristique importante du jeu d'instructions de tout microprocesseur et en particulier du 6502 où un effort particulier a été fait pour le munir du plus grand nombre possible d'instructions à un seul octet afin d'améliorer l'efficacité des programmes à l'exécution. Toutefois, le fait de se contenter d'une longueur de mots de 8 bits a entraîné d'importantes limitations qui seront esquissées dans la suite. C'est un exemple classique du compromis entre la vitesse d'exécution et la souplesse de programmation. La représentation binaire des instructions est dictée par le constructeur et leur liste est donnée à la fin du livre. Tout programme s'exprimera comme suite de ces instructions binaires. Le codage binaire proprement dit des instructions du 6502 est présenté au chapitre IV.

2 — Représentation des données numériques

La représentation des nombres n'est pas si évidente et il faut distinguer plusieurs cas. Il faut d'abord représenter des entiers. Il faut aussi représenter des nombres signés c'est-à-dire positifs ou négatifs et, enfin, nous devons être capables de représenter des nombres décimaux. Passons en revue ces différents impératifs et les solutions possibles.

La représentation des entiers peut se faire sous forme *binaire directe*. La représentation binaire directe n'est autre que la représentation de la valeur d'un nombre dans le système de numération binaire. Dans le système binaire, le bit le plus à droite représente 2 à la puissance 0. Le suivant immédiatement à gauche représente 2 à la puissance 1, le suivant 2 à la puissance 2 et le bit le plus à gauche représente 2 à la puissance 7 = 128 :

$$b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0$$

représente

$$b_7\, 2^7 + b_6\, 2^6 + b_5\, 2^5 + b_4\, 2^4 + b_3\, 2^3 + b_2\, 2^2 + b_1\, 2^1 + b_0\, 2^0$$

Les puissances de 2 sont :

$2^7 = 128$; $2^6 = 64$; $2^5 = 32$; $2^4 = 16$; $2^3 = 8$; $2^2 = 4$; $2^1 = 2$; $2^0 = 1$.

La représentation binaire est analogue à la représentation décimale des nombres où, par exemple, 123 représente :

$$\begin{aligned} & 1 \times 100 = 100 \\ + \; & 2 \times 10 = 20 \\ + \; & 3 \times 1 = 3 \\ & = 123 \end{aligned}$$

Notez que $100 = 10^2$; $10 = 10^1$; $1 = 10^0$.

Dans cette « numération de position », chaque chiffre représente une puissance de 10. Dans le système binaire, chaque chiffre binaire, ou « bit », représente une puissance de 2 (et non une puissance de 10 comme dans le système décimal).

Exemple : « 00001001 » en binaire, représente :

$1 \times 1 = 1$ (2^0)
$0 \times 2 = 0$ (2^1)
$0 \times 4 = 0$ (2^2)
$1 \times 8 = 8$ (2^3)
$0 \times 16 = 0$ (2^4)
$0 \times 32 = 0$ (2^5)
$0 \times 64 = 0$ (2^6)
$0 \times 128 = 0$ (2^7)

= en décimal : 9

Examinons d'autres exemples :

« 10000001 » représente :

$1 \times 1 = 1$
$0 \times 2 = 0$
$0 \times 4 = 0$
$0 \times 8 = 0$
$0 \times 16 = 0$
$0 \times 32 = 0$
$0 \times 64 = 0$
$1 \times 128 = 128$

soit en décimal : 129

Par suite « 10000001 » représente le nombre décimal 129.

En examinant la représentation binaire des nombres, vous allez comprendre pourquoi les bits sont numérotés de 0 à 7, de droite à gauche. Le bit 0 est « b_0 » et il correspond à 2^1. Le bit 1 est « b_1 » et il correspond à 2^1 et ainsi de suite.

Les équivalents binaires des nombres de 0 à 255 sont montrés Figure 1-2.

Exercice 1.1 : *Quelle est la valeur décimale de « 11111110 » ?*

CONVERSION DECIMAL-BINAIRE

Inversement, calculons l'équivalent binaire de « 11 » en décimal :

$11 \div 2 = 5$ reste $1 \rightarrow 1$ bit de moins significatif (LSB)
$5 \div 2 = 2$ reste $1 \rightarrow 1$
$2 \div 2 = 1$ reste $0 \rightarrow 0$
$1 \div 2 = 0$ reste $1 \rightarrow 1$ bit le plus significatif (MSB)

DECIMAL	BINAIRE	DECIMAL	BINAIRE
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010		
3	00000011		
4	00000100		
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000		
9	00001001		
10	00001010	127	01111111
11	00001011	128	10000000
12	00001100	129	10000001
13	00001101		
14	00001110		
15	00001111		
16	00010000		
17	00010001		
		254	11111110
31	00011111	255	11111111

Fig. 1-2 : Table de conversion binaire-décimal

L'équivalent binaire est 1011 (on lit de droite à gauche de bas en haut).

L'équivalent binaire d'un nombre décimal s'obtient en divisant par deux successivement jusqu'à obtenir un quotient 0.

Exercice 1.2 : *Quel est l'équivalent binaire de 257 ?*

Exercice 1.3 : *Convertissez 19 en binaire puis à nouveau en décimal.*

OPERATIONS EN BINAIRE

Les règles de l'arithmétique binaire sont simples. Les règles d'addition sont :

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = (1) 0

où (1) représente une retenue (notez que « 10 » est la représentation de « 2 » décimal). La soustraction s'effectuera en « ajoutant le com-

plément » ce qui sera expliqué lorsque nous aurons appris comment représenter les nombres négatifs.

Exemple :

```
   (2)      10
 + (1)    + 01
 -----    -----
 = (3)    = 11
```

L'addition s'effectue exactement comme en décimal en ajoutant les colonnes de droite à gauche :

Addition de la colonne de droite :

```
    1 0
  + 0 1
  -----
(0 + 1 =   1 ; pas de retenue)
```

Addition de la colonne suivante :

```
  1 0
+ 0 1
-----
  1 1   (1 + 0 = 1 ; pas de retenue)
```

Exercice 1.4 : *Calculez 5 + 10 en binaire. Vérifiez que le résultat est 15.*

Exemples supplémentaires d'addition binaire :

```
   0010  (2)        0011  (3)
 + 0001  (1)      + 0001  (1)
 ---------        ---------
 = 0011  (3)      = 0100  (4)
```

Le dernier exemple illustre le rôle de la retenue.

Regardons les bits les plus à droite : 1 + 1 = (1) 0.

On génère une retenue de 1 qui doit être ajoutée aux bits suivants :

```
    001 — la colonne 0 vient d'être traitée
+   000 —
+     1   retenue
-------
= (1) 0 — où (1) indique qu'une nouvelle retenue apparaît en
          colonne 2.
```

Le résultat final est 0100.

Autre exemple :

```
  0111       ( 7)
+ 0011     + ( 3)
------     -----
= 1010     = (10)
```

Ici encore, une retenue apparaît qui se propage jusqu'à la colonne la plus à gauche.

Exercice 1.5 : *Calculez le résultat de :*

$$\begin{array}{r} 1111 \\ +\ 0001 \\ \hline =\ \ ? \end{array}$$

Le résultat tient-il en 4 bits ?

Il est donc possible avec 8 bits de représenter directement les nombres 00000000 à 11111111 c'est-à-dire 0 à 255. Deux obstacles apparaissent immédiatement. D'abord nous ne représentons que des nombres positifs. Ensuite, la grandeur de ces nombres est limitée à 255 si nous ne prenons que 8 bits. Traitons chacun de ces problèmes à son tour.

Représentation binaire signée

En représentation binaire signée, le bit le plus à gauche indique le signe du nombre. Traditionnellement, on utilise « 0 » pour signaler un nombre *positif* et « 1 » pour dénoter un nombre *négatif*. Maintenant, « 1111 1111 » va représenter — 127 tandis que « 0111 1111 » va représenter + 127. Nous pouvons maintenant représenter des nombres positifs ou négatifs mais la taille maximum de nos nombres s'est réduite à 127.

Exemple : « 0000 0001 » représente + 1 (le zéro en tête veut dire « + », suivi par « 000 0001 » = 1).

« 1000 0001 » représente — 1 (le « 1 » en tête veut dire « — »).

Exercice 1.6 : *Quelle est la représentation de « — 5 » en binaire signé ?*

Occupons-nous maintenant du problème de la grandeur des nombres : pour représenter des nombres plus grands, il faut faire appel à un plus grand nombre de bits. Par exemple, si l'on utilise 16 bits (2 octets) pour représenter les nombres, on pourra admettre des nombres allant de — 32 K à + 32 K en représentation binaire signée (dans le jargon informatique, 1 K signifie 1024). Si cette grandeur est encore trop faible on utilisera 3 octets ou plus. Si on souhaite représenter des entiers très grands, il est nécessaire d'utiliser un grand nombre d'octets pour leur représentation interne. C'est pourquoi la plupart des BASIC simples, comme d'ailleurs d'autres langages, ne fournissent qu'une précision limitée pour les entiers. De cette manière, ils peuvent utiliser un format assez court pour les entiers qu'ils ont à manipuler. Les meilleures versions de BASIC ou des autres langages fournissent un plus grand nombre de chiffres significatifs au prix d'un plus grand nombre d'octets consacré à chaque nombre.

Occupons-nous maintenant d'un autre problème, celui de la vitesse de calcul. Nous allons essayer de faire une addition en représentation binaire signée telle que nous l'avons introduite.

Ajoutons « — 5 » à « + 7 ».

+ 7 a pour représentation 00000111
— 5 a pour représentation 10000101
La somme est .....10001100 ou — 12

Ce n'est pas le résultat correct. Celui-ci devrait être + 2. En d'autres termes, l'addition binaire ne fonctionne pas correctement pour les nombres signés. C'est bien ennuyeux car il est évident que l'ordinateur ne doit pas seulement représenter l'information : il doit aussi effectuer des opérations arithmétiques sur elle.

La solution de ce problème est la représentation en *complément à deux* qu'on utilisera au lieu de la représentation *binaire signée*. Pour introduire les compléments à deux, nous introduisons une étape intermédiaire : *les compléments à un.*

Complément à un

Dans la représentation en complément à un, tous les entiers positifs sont représentés en format binaire correct. Par exemple « + 3 » a pour représentation 00000011 comme d'habitude. Mais son opposé « — 3 » est obtenu en complémentant chaque bit de l'original. Chaque 0 est transformé en 1 et chaque 1 en 0. Dans notre exemple la représentation en complément à un de « — 3 » sera 11111100.

Autre exemple :
+ 2 s'écrit 00000010
— 2 s'écrit 11111101.

Notez que, dans cette représentation, les nombres positifs commencent pas un « 0 » à gauche ; les nombres négatifs, par un « 1 ».

Exercice 1.7 : *La représentation de + 6 est 00000110. Quelle est la représentation de − 6 en complément à 1 ?*

A titre de test, ajoutons − 4 à + 6.

— 4 s'écrit 11111011
+ 6 s'écrit 00000110
La somme est ..... (1) 00000001 où (1) indique une retenue.

C'est-à-dire « 1 » avec une retenue.

Le résultat « correct » devrait être « 2 » ou « 00000010 ».

Essayons encore :

– 3 s'écrit	11111100
– 2 s'écrit	11111101
La somme est ..... (1)	11111001

soit « 1 » plus une retenue. Le résultat correct devrait être « – 5 ». La représentation de « – 5 » est 11111010. L'addition n'a pas marché.

Cette représentation permet de présenter les nombres positifs ou négatifs. Mais une addition ordinaire ne s'effectue pas correctement. Il nous faut encore une autre représentation. Elle est déduite du complément à un et s'appelle le complément à deux.

Représentation en Complément à Deux

En complément à deux, les nombres positifs ont exactement la même représentation habituelle qu'en complément à un. La différence réside dans la représentation des *nombres négatifs.* La représentation en complément à deux d'un nombre négatif s'obtient en calculant d'abord le complément à un, et en lui *ajoutant un* ensuite. Voyons le sur un exemple : + 3 a pour représentation 00000011. Son complément à un est 11111100. Le complément à 2 s'obtient en ajoutant 1, d'où 11111101.

Essayons une addition :

(3)	00000011
+ (5)	+ 00000101
= (8)	= 00001000

Le résultat est correct.

Essayons d'ajouter un nombre négatif :

(3)	00000011
(– 5)	+ 11111011
	= 11111110

Identifions le résultat en calculant son complément à deux :

le complément à un de 11111110 est 00000001
ajoutons 1 + 1
le complément à deux est donc 00000010 ou + 2.

Donc notre résultat ci-dessus « 11111110 » représente « – 2 ». Le résultat est correct.

Nous avons maintenant essayé l'addition et la soustraction (effectuée en ajoutant le complément à deux) et les résultats ont été corrects (en ignorant la retenue). Il semble que le complément à deux marche !

Exercice 1.8 : *Quelle est la représentation de + 127 en complément à deux ?*

Exercice 1.9 : *Quelle est la représentation de − 128 en complément à deux ?*

Ajoutons maintenant + 4 et − 3 :

+ 4 s'écrit 00000100
− 3 s'écrit 11111101

La somme est ..... (1) 00000001 où (1) indique la retenue.

Cette fois, le résultat est correct, y compris le signe (on ne tient pas compte de la retenue). Le complément à deux marche ! Sans en donner la démonstration mathématique complète, nous affirmons que cela marche : en complément à deux, il est possible d'ajouter ou de soustraire des nombres sans s'occuper de leur signe. On utilise les règles habituelles d'addition et de soustraction en binaire et on obtient le bon résultat y compris le signe. On ne tient pas compte de la retenue. C'est un avantage considérable. Si ce n'était pas le cas, il faudrait rectifier le signe à chaque résultat ce qui entraînerait des temps d'addition et de soustraction bien plus lents.

Pour être complets, disons que le complément à deux est simplement la représentation la plus commode à utiliser avec les processeurs simples comme les microprocesseurs. On peut utiliser d'autres représentations sur les processeurs complexes. Par exemple, on peut utiliser le complément à un, mais il faut des circuits spéciaux pour « corriger les résultats ».

On supposera dans toute la suite, que tous les entiers signés seront représentés en complément à deux. Cf. Figure 1-3 pour une table des compléments.

Exercice 1.10 : *Quels sont le plus petit et le plus grand nombres que l'on peut représenter en complément à deux avec un seul octet ?*

Exercice 1.11 : *Calculez le complément à 2 de 20. Puis calculez le complément à 2 de votre résultat. Est-ce que vous retrouvez 20 ?*

Les exemples qui suivent vont servir à démontrer les règles de complément à deux. En particulier C dénote la condition éventuelle de retenue (ou emprunt) : c'est le bit 8 du résultat.

V représente un débordement du complément à deux, c'est-à-dire quand le signe du résultat est changé « accidentellement » parce que les nombres sont trop grands. C'est une retenue interne du bit 6 sur le bit 7 (bit de signe). Ceci sera clarifié ci-dessous.

Voyons maintenant le rôle de la retenue « C » et du débordement « V ».

+	CODE EN COMPLEMENT A 2	−	CODE EN COMPLEMENT A 2
+ 127	01111111	− 128	10000000
+ 126	01111110	− 127	10000001
+ 125	01111101	− 126	10000010
		− 125	10000011
+ 65	01000001	− 65	10111111
+ 64	01000000	− 64	11000000
+ 63	00111111	− 63	11000001
+ 33	00100001	− 33	11011111
+ 32	00100000	− 32	11100000
+ 31	00011111	− 31	11100001
+ 17	00010001	− 17	11101111
+ 16	00010000	− 16	11110000
+ 15	00001111	− 15	11110001
+ 14	00001110	− 14	11110010
+ 13	00001101	− 13	11110011
+ 12	00001100	− 12	11110100
+ 11	00001011	− 11	11110101
+ 10	00001010	− 10	11110110
+ 9	00001001	− 9	11110111
+ 8	00001000	− 8	11111000
+ 7	00000111	− 7	11111001
+ 6	00000110	− 6	11111010
+ 5	00000101	− 5	11111011
+ 4	00000100	− 4	11111100
+ 3	00000011	− 3	11111101
+ 2	00000010	− 2	11111110
+ 1	00000001	− 1	11111111
+ 0	00000000	− 0	00000000

Fig. 1-3 : Table des compléments

La retenue C

Voici un exemple de retenue :

$$\begin{array}{rcl} (128) & & 10000000 \\ +\ (129) & & +\ 10000001 \\ \hline (257) & = & (1)\ 00000001 \end{array}$$

où (1) dénote la retenue.

Le résultat nécessite un neuvième bit (bit « 8) puisqu'on numérote à partir de 0). C'est le bit de retenue.

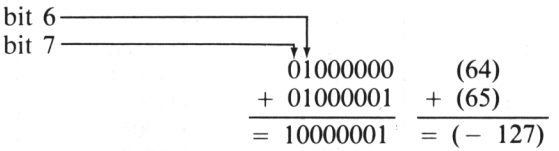
Si nous supposons que la retenue est le 9e bit du résultat, nous interprétons le résultat comme 100000001 = 257.

Mais il faut déceler la retenue et la manipuler avec soin. A l'intérieur du microprocesseur, les registres utilisés pour garder les informations ont généralement 8 bits seulement. Lorsqu'on rangera le résultat, seuls les bits 0 à 7 seront préservés.

Par suite, une retenue nécessite toujours une action spéciale : elle doit être détectée par des instructions spéciales puis traitée. La traiter conduit soit à la stocker quelque part (avec une instruction spéciale) ou à l'ignorer ou à décider qu'il y a une erreur (si le résultat le plus grand permis est « 11111111 »).

Le débordement (ou dépassement de capacité) V

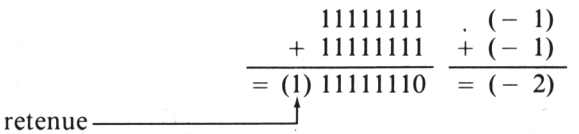
Voici un exemple de débordement :



Une retenue interne a été générée du bit 6 sur le bit 7. Cela s'appelle un débordement.

Le résultat est devenu négatif « par accident ». Cette situation doit être détectée pour pouvoir être corrigée.

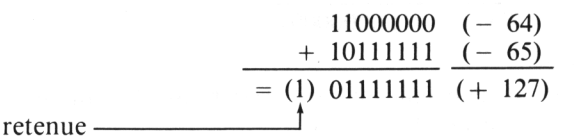
Regardons un autre cas :



Dans ce cas, il y a eu une retenue du bit 6 vers le bit 7, mais aussi du bit 7 vers le bit 8 (la véritable retenue C comme nous l'avons vu à la section précédente). Les règles de l'arithmétique en complément à deux spécifient qu'on doit ignorer cette retenue. Le résultat est alors correct.

C'est parce que la retenue du bit 6 vers le bit 7 n'a pas modifié le bit de signe.

Ce n'est pas une condition de *débordement*. Lorsqu'on opère sur des nombres négatifs, le débordement n'est pas une simple retenue du bit 6 sur le bit 7. Examinons encore un exemple :



Cette fois, il n'y a pas de retenue interne du bit 6 vers le bit 7, mais il y a eu une retenue externe. Le résultat est incorrect, car le bit 7 s'est trouvé changé. Il doit y avoir une condition de débordement indiquée.

Le débordement se produit dans quatre cas :

1 — addition de grands nombres positifs
2 — addition de grands nombres négatifs
3 — soustraction d'un grand nombre positif à un grand nombre négatif.
4 — soustraction d'un grand nombre négatif à un grand nombre positif.

Techniquement, l'indicateur de débordement, bit spécial réservé à cet usage et appelé « indicateur d'état » sera mis à 1 lorsqu'il y a une retenue du bit 6 vers le bit 7 et pas de retenue externe, ou bien lorsqu'il n'y a pas de retenue du bit 6 vers le bit 7 mais qu'il y en a une à l'extérieur. Ceci indique que le bit 7 (bit de signe du résultat) a été changé accidentellement. Pour le lecteur qui s'intéresse à la technique, disons que l'indicateur de débordement est établi en faisant le ou exclusif de la retenue qui arrive au bit 7 et de celle qui en sort. Pratiquement tous les microprocesseurs disposent d'un indicateur de débordement qui détecte cette condition car elle nécessite une action corrective.

Le débordement signifie que le résultat d'une addition nécessite plus de bits qu'on n'en dispose dans le registre standard de 8 bits utilisé pour contenir le résultat.

Retenue et débordement

Le bit de retenue et le bit de débordement sont appelés des « Indicateurs d'état ». Ils sont présents dans tout microprocesseur et nous apprendrons à les utiliser effectivement au chapitre suivant. Ces deux indicateurs sont situés dans un registre spécial appelé « registre d'état ». Ce registre contient d'autres indicateurs dont la fonction sera expliquée au Chapitre IV.

Exemples

Illustrons maintenant le fonctionnement de la retenue et du débordement sur des exemples concrets. Dans ces exemples, le symbole V désigne le débordement et C la retenue.

S'il n'y a pas eu de débordement, V = 0. S'il y en a eu, V = 1. (De même pour la retenue C.) Rappelez-vous que d'après les règles de l'arithmétique en complément à 2, on ignore la retenue (nous l'admettons ici sans démonstration mathématique).

Positif + positif :

```
  00000110  (+6)
+ 00001000  (+8)
= 00001110  (+14) V = 0 C = 0
```

(EXACT)

Positif + positif avec débordement

```
+ 01111111  (+127)
+ 00000001  (+1)
= 10000000  (—128) V = 1 C = 0
```

(FAUX) (car il y a eu débordement)

Positif + négatif (résultat positif)

```
  00000100  (+4)
+ 11111110  (—2)
= 00000010  (+2) V = 0 C = 1(ignorée)
```

(EXACT)

Positif + négatif (résultat négatif)

```
  00000010  (+2)
+ 11111100  (—4)
= 11111110  (—2) V = 0 C = 0
```

(EXACT)

Négatif + négatif avec débordement

```
    10000001  (— 127)
  + 11000010  (— 62)
= (1) 01000011  (67)      V = 1  C = 1
```

(FAUX)

Cette fois, il y a en sous-dépassement de capacité en ajoutant deux grands nombres négatifs. Le résultat devrait être — 189 ce qui est trop grand pour tenir en 8 bits.

Exercice 1.12 : *Complétez les additions suivantes. Indiquez le résultat, la retenue C, le débordement V et dites si le résultat est correct ou non.*

```
  10111111  (__)                11111010  (__)
+ 11000001  (__)              + 11111001  (__)
=           V =     C =       =           V =     C =
```

☐ EXACT ☐ FAUX ☐ EXACT ☐ FAUX

```
  00010000  (__)                 01111110  (__)
+ 01000000  (__)               + 00101010  (__)
=           V =     C =        =           V =     C =
```

☐ EXACT ☐ FAUX ☐ EXACT ☐ FAUX

Exercice 1.13 : *Pouvez-vous montrer un exemple de débordement obtenu en ajoutant un nombre positif et un nombre négatif ? Pourquoi ?*

REPRESENTATION EN FORMAT FIXE

Nous savons maintenant représenter les entiers signés. Mais nous n'avons pas encore résolu le problème de la grandeur des nombres. Si nous voulons représenter des entiers plus grands, nous aurons besoin de plusieurs octets. Pour effectuer les opérations arithmétiques efficacement il est nécessaire d'utiliser un nombre d'octets fixe et non variable. Par suite, une fois le nombre d'octets choisi, par exemple deux ou trois, la valeur absolue maximum des nombres qui peuvent être représentés est déterminée.

Exercice 1.14 : *Quels sont le plus grand et le plus petit nombres qui peuvent être représentés en complément à deux avec deux octets ?*

LE PROBLEME DE LA GRANDEUR DES NOMBRES

Lorsque nous avons fait des additions, nous nous sommes restreints à 8 bits parce que le processeur que nous utiliserons ne traite intérieurement que 8 bits à la fois. Mais cela nous restreint à des nombres compris entre – 128 et + 127. Manifestement, c'est insuffisant pour beaucoup d'applications.

On utilisera la multiple précision pour augmenter le nombre de chiffres capables d'être représentés. On peut alors utiliser un format à 2 octets, 3 octets ou N octets. Par exemple, examinons le format « double précision » sur 16 bits :

```
00000000 00000000  représente  « 0 »
00000000 00000001  représente  « 1 »
01111111 11111111  représente  « 32767 »
11111111 11111111  représente  « – 1 »
11111111 11111110  représente  « – 2 »
```

Exercice 1.15 : *Quel est le plus grand entier négatif qu'on peut représenter en complément à deux en format triple précision ?*

Néanmoins, cette méthode a des inconvénients. Lorsqu'on ajoute deux nombres, par exemple, il faut généralement le faire 8 bits à la fois. Ceci sera expliqué au chapitre III (techniques de base de la programmation). Cela ralentit le traitement. De plus, cette représentation utilise 16 bits pour n'importe quel nombre même s'il pourrait tenir en 8 bits. Par suite, on utilise souvent 16 bits ou 32 bits mais rarement plus.

Considérons un point important : quelque soit le nombre de bits N choisi pour la représentation, il est fixé. Si jamais un calcul intermédiaire génère un résultat exigeant plus que N bits, il y aura des bits de perdus. Le programme conserve généralement les N bits les plus à gauche (les plus significatifs) et il laisse tomber les plus bas. Cela s'appelle « troncation des résultats ».

Voici un exemple en décimal, avec une représentation à 6 chiffres :

```
   123456
×      1,2
---------
   246912
  123456
---------
= 148147,2
```

Le résultat exige 7 chiffres ! Le 2 qui suit la virgule sera abandonné et le résultat final sera 148147. Il a été tronqué. D'habitude, tant que la position de la virgule n'est pas perdue, on utilise cette méthode pour étendre la gamme des opérations possibles, au détriment de la précision.

Le problème est le même en binaire. Les détails d'une multiplication binaire seront donnés au chapitre III.

Malheureusement, si l'on utilise une représentation des nombres à précision fixée, on n'obtiendra que des résultats *approchés* lorsqu'on effectuera des multiplications ou des divisions (les résultats intermédiaires seront tronqués). On peut avoir une précision excellente qui sera suffisante pour les calculs mathématiques usuels.

Malheureusement, en comptabilité, aucune perte de précision n'est tolérable. Par exemple, si un client achète un grand nombre d'articles identiques qui seront totalisés sur une caisse enregistreuse il ne sera pas acceptable d'avoir un montant sur un chiffre défini à un dollar près. Il faut utiliser une autre représentation dès que la précision du résultat est essentielle. La solution normalement utilisée est le « DCB » ou décimal codé binaire.

La représentation DCB

Le principe de la représentation DCB est de coder chaque chiffre décimal séparément et d'utiliser autant de bits que nécessaire pour

représenter toute l'information. Pour représenter chacun des chiffres de 0 à 9, il faut quatre bits. En effet, trois bits ne donneraient que huit combinaisons et donc ne pourraient pas permettre de coder dix chiffres. Quatre bits permettent seize combinaisons et sont par suite suffisants pour coder les chiffres « 0 » à « 9 ». On peut aussi remarquer que six des combinaisons possibles sont inutilisées en représentation DCB. Cela entraîne un problème potentiel lors des additions et des soustractions, qu'il faudra résoudre. Comme on n'a besoin que de quatre bits pour coder un chiffre DCB, on peut en mettre deux dans un octet. Cela s'appelle le « DCB compact ».

Par exemple, « 00000000 » représente « 00 » en DCB. « 10011001 » représente « 99 ».

CODE	SYMBOLE DCB	CODE	SYMBOLE DCB
0000	0	1000	8
0001	1	1001	9
0010	2	1010	inutilisé
0011	3	1011	inutilisé
0100	4	1100	inutilisé
0101	5	1101	inutilisé
0110	6	1110	inutilisé
0111	7	1111	inutilisé

Fig. 1-4 : Table DCB

Un code DCB se lit comme suit :

0010 0001

chiffre DCB « 2 »
chiffre DCB « 1 »

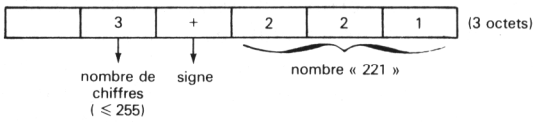
d'où le nombre « 21 ».

Exercice 1.16 : *Quelles sont les représentations DCB de « 29 » et de « 91 » ?*

Exercice 1.17 : *« 10100000 » est-elle une représentation DCB correcte ? Pourquoi ?*

On utilise autant d'octets que nécessaire pour représenter tous les chiffres DCB donnés. Généralement, on utilise un ou plusieurs quartets en tête de la représentation pour indiquer le nombre total de quartets, c'est-à-dire le nombre de chiffres DCB utilisés. Un autre quartet (ou un octet) est utilisé pour indiquer la position de la virgule. Toutefois, les conventions peuvent varier.

Voici un exemple de représentation DCB d'entiers sur plusieurs octets :



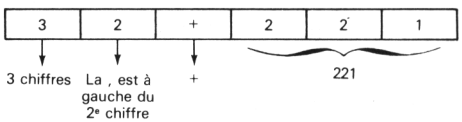
Ceci représente + 221.

(Le signe peut, par exemple, être représenté par 0000 pour + et 0001 pour −).

Exercice 1-18 : *A l'aide de la même convention, représentez « − 23123 ». Montrez-le en format BCD comme ci-dessous, puis en binaire.*

Exercice 1-19 : *Montrez la représentation BCD de « 222 » et de « 221 », puis du produit 222 × 111. (Calculez le résultat à la main puis montrez-le dans la représentation ci-dessous.)*

La représentation DCB peut facilement s'appliquer aux nombres fractionnaires. Par exemple + 2,21 peut être représenté sous la forme :



L'avantage du DCB est de fournir des résultats absolument exacts. Son inconvénient est d'utiliser une grande quantité de mémoire entraînant des opérations arithmétiques lentes. Ce n'est acceptable que dans un environnement de comptabilité et n'est normalement pas utilisé dans les autres cas.

Exercice 1-20 : *Combien de bits sont nécessaires pour représenter « 9999 » en DCB ? Et en binaire, complément à deux ?*

Nous avons maintenant résolu les problèmes liés à la représentation des entiers, des entiers signés, même très grands. Nous avons même déjà présenté une méthode possible de représenter les nombres fractionnaires dans le cas du DCB. Examinons maintenant le problème de la représentation des nombres fractionnaires dans le cas général.

Représentation en virgule flottante

Le principe de base est que les nombres fractionnaires doivent être représentés suivant un format fixe. Pour ne pas gaspiller de bits, la représentation *normalise* tous les nombres.

Par exemple : « 0.000123 » gâche trois zéros sur la gauche du nombre : ils n'ont d'autre signification que d'indiquer la position de la virgule. La normalisation de ce nombre le transformerait en 0,123 $\times$ 10^{-3}. « 123 » s'appelle *mantisse normalisée.* « — 3 » est *l'exposant.* On a normalisé le nombre en éliminant tous les zéros non significatifs à gauche du nombre et en ajustant l'exposant.

Considérons un autre exemple :

22,1 se normalise en 0,221 $\times$ 10^{+2}

ou M $\times$ 10^{2} où M est la mantisse, et E est l'exposant.

On peut voir facilement qu'un nombre normalisé est caractérisé par une mantisse comprise entre 0,1 et 1 dans tous les cas où le nombre n'est pas zéro. En d'autres termes cela peut être résumé mathématiquement sous la forme :

$0{,}1 \leqslant M < 1$ ou $10^{-1} \leqslant M < 10^{0}$.

De façon analogue, en représentation binaire :

$2^{-1} \leqslant M < 2^{0}$

Où M est la valeur absolue de la mantisse (on ignore le signe).

Par exemple :

111,01 se normalise en : 0,11101 $\times$ 2^{3}.

La mantisse est 11101 ; l'exposant est + 3.

Nous avons défini le principe de la représentation. Examinons maintenant le format réel. La représentation flottante type apparaît ci-dessus.

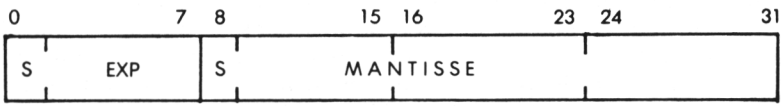


Fig. 1-5 : Représentation virgule flottante type

Dans la représentation de cet exemple, on utilise 4 octets soit un total de 32 bits. Le premier octet, sur la gauche de l'illustration sert à représenter l'exposant. Tant l'exposant que la mantisse sont représentés en complément à deux. Il en résulte que l'exposant maximum est 2^{127}, et le plus petit, 2^{-128}.

On utilise trois octets pour la mantisse. Comme le premier bit indique le signe en complément à deux, cela laisse 23 bits pour représenter la valeur absolue de la mantisse.

Exercice 1-21 : *Combien de chiffres décimaux peuvent être représentés avec une mantisse sur 23 bits ?*

Ceci n'est qu'un exemple possible de représentation flottante. Il est possible d'utiliser seulement trois octets comme il est possible d'en utiliser plus. La représentation sur quatre octets proposée ci-dessus n'en est qu'une très répandue qui représente un compromis raisonnable entre la précision, la grandeur des nombres, l'encombrement mémoire et l'efficacité pour les calculs.

Nous avons maintenant exploré les problèmes liés à la représentation des nombres et nous savons représenter sous forme entière, signée ou fractionnaire. Examinons maintenant la représentation interne des données alphanumériques.

Représentations des données alphanumériques

La représentation des données alphanumériques, c'est-à-dire de caractères est tout à fait immédiate : chaque caractère est codé en un code de 8 bits. Il n'y a que deux codes répandus dans le monde des ordinateurs, l'ASCII et l'EBCDIC. ASCII signifie « Code Standard Américain pour l'échange d'information » ; il est universellement employé dans le monde des microprocesseurs. L'EBCDIC est une variante de l'ASCII utilisée par IBM ; il n'est donc pas utilisé par les microprocesseurs sauf si on veut s'interfacer à un terminal IBM. Examinons brièvement le codage ASCII. Nous devons coder les 26 lettres de l'alphabet, majuscules et minuscules, plus 10 chiffres, plus, peut-être, 20 symboles spéciaux supplémentaires. Cela peut être facilement réalisé avec 7 bits qui autorisent 128 codes possibles. Tous les caractères sont donc codés sur 7 bits. Le 8e bit lorsqu'il est utilisé, est le bit de parité. La parité est une technique pour vérifier que le contenu d'un mot n'a pas été changé accidentellement. On compte le nombre de uns dans le mot et on met le 8e bit à 1 si le compte trouvé était impair, ce qui rend le total pair. Cela s'appelle de la parité paire. On peut aussi utiliser la parité impaire, c'est-à-dire déterminer le bit le plus à gauche pour que le total de 1 soit impair.

Exemple : Calculons le bit de parité pour « 0010011 » en parité paire. Le nombre de uns est 3. Le bit de parité doit donc être à 1 de façon que le nombre total de 1 soit 4 c'est-à-dire pair. Le résultat est 10010011 où le 1 en tête est le bit de parité et 0010011 identifie le caractère.

La table des codes ASCII 7 bits est montrée ci-dessous. Elle est, en pratique, utilisée « telle quelle », c'est-à-dire sans parité en ajou-

tant un 0 à gauche, ou alors avec parité en ajoutant à gauche le bit voulu.

Exercice 1-22 : *Calculez la représentation 8 bits des chiffres « 0 » à « 9 » en parité paire. (Ces codes serviront dans des exemples d'applications du chapitre 8.)*

Exercice 1-23 : *Même chose pour les lettres « A » à « F ».*

HEX		0	1	2	3	4	5	6	7
	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

Fig. 1-6 : Table de conversion ASCII

En outre, dans certaines situations spécialisées comme les télécommunications, on peut utiliser d'autres codages supplémentaires comme les codes correcteur d'erreurs. Ils vont au-delà du sujet de ce livre.

Nous avons appris comment représenter à la fois le programme et les données à l'intérieur de l'ordinateur. Examinons maintenant les représentations externes possibles.

REPRESENTATION EXTERNE DE L'INFORMATION

La représentation externe de l'information est la manière dont l'information est présentée à *l'utilisateur* c'est-à-dire généralement

au programmeur. L'information est présentée à l'extérieur essentiellement sous trois formes : binaire, octal ou hexadécimal, et symbolique.

1 — Binaire

On a vu que l'information est rangée à l'intérieur de la machine sous forme de bits (séquences de zéros et de uns). Il est quelquefois désirable d'afficher cette information interne directement sous forme binaire : c'est appelé *représentation interne.* Un exemple simple est fourni par les voyants du panneau de la commande du micro-ordinateur (si toutefois il y a encore un panneau de commande). Dans le cas d'un microprocesseur 8 bits, le panneau de commande est généralement muni de 8 LED qui permettent d'afficher le contenu de n'importe quel registre. Une LED allumée dénote un 1, une LED éteinte dénote 0.

Une telle représentation binaire peut être nécessaire pour le dépannage fin d'un programme complexe, spécialement s'il fait intervenir des entrées-sorties, mais elle est bien sûr impraticable à l'échelle humaine. C'est parce que, dans la plupart des cas, on préfère examiner l'information sous forme symbolique. Ainsi « 9 » est bien plus facile à se rappeler ou à comprendre que « 1001 ». Des représentations plus commodes ont été développées qui facilitent la communication homme-machine.

2 — Octal et Hexadécimal

En octal et en hexadécimal respectivement, on représente trois ou quatre bits par un seul symbole. Dans le système octal, toute combinaison de trois bits est représentée par un nombre compris entre 0 et 7.

Binaire	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Fig. 1-7 : Symboles de l'octal

Par exemple, « 00 100 100 » en binaire est représenté par :
0 4 4

soit « 044 » en octal.

Autre exemple : « 11 111 111 » s'écrit :
3 7 7

ou « 377 » en octal.

Inversement, l'octal « 211 » représente 10001001 en binaire.

L'octal était utilisé traditionnellement dans les anciens ordinateurs qui avaient des mots allant de 6 à, peut-être, 60 bits.

Plus récemment, avec la domination des microprocesseurs 8 bits, le format 8 bits est devenu le standard, et une autre représentation plus pratique est utilisée. C'est l'hexadécimal.

En hexadécimal, on représente chaque groupe de 4 bits par un chiffre hexadécimal. Les chiffres hexadécimaux sont les nombres de 0 à 9 puis les lettres A, B, C, D, E et F. Par exemple « 0000 » est représenté par « 0 ». « 0001 » est représenté par « 1 » et « 1111 » est représenté par la lettre F (cf. Fig. 1-8).

DECIMAL	BINAIRE	HEXADECIMAL	OCTAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Fig. 1-8 : Codes hexadécimaux

Exemple : 1010 0001 en binaire, est représenté par :
A 1 en hexadécimal.

Exercice 1.25 : *Quelle est la représentation hexadécimale de « 10101010 » ?*

Exercice 1.26 : *Réciproquement, quel est l'équivalent binaire de l'hexadécimal « FA » ?*

Exercice 1.27 : *Comment s'écrit « 01000001 » en octal ?*

L'hexadécimal offre l'avantage de représenter un octet avec seulement deux chiffres. C'est plus facile à visualiser ou à mémoriser, et plus facile à taper à la machine. Par suite l'hexadécimal est la méthode de prédilection pour représenter les groupes de bits sur tous les nouveaux microordinateurs. Naturellement, quand l'information présente en mémoire a une signification définie telle que représentation de textes ou de nombres, l'hexadécimal ne convient plus pour figurer cette information. Dans ces cas-là, on utilise une troisième méthode.

3 — Représentation symbolique

La représentation symbolique désigne la représentation externe de l'information dans sa forme symbolique proprement dite. Par exemple, les nombres décimaux sont représentés comme nombres décimaux et non comme séquence de bits ou de symboles hexadécimaux. De même les textes sont représentés en clair. Bien sûr, la représentation symbolique est la plus pratique pour l'utilisateur. On s'en sert à chaque fois qu'un périphérique de visualisation approprié est disponible, comme un écran cathodique ou une imprimante. Malheureusement dans les systèmes plus petits comme les microordinateurs sur une carte, il est trop onéreux de fournir de tels périphériques et l'utilisateur doit se contenter de communiquer avec l'ordinateur en hexadécimal.

Résumé des représentations externes

La représentation *symbolique* de l'information est la plus désirable puisqu'elle est la plus naturelle pour l'utilisateur humain. Néanmoins, elle nécessite un interface coûteux sous forme d'un clavier alphanumérique plus une imprimante ou un terminal à écran. Pour cette raison, on n'en disposera probablement pas avec les systèmes de plus bas prix. On utilisera une autre représentation et, dans ce cas, l'hexadécimal domine. Ce n'est que dans ces rares cas liés au dépannage fin au niveau software ou hardware que l'on utilise la représentation binaire. Le *binaire* affiche directement le contenu des registres ou des mémoires sous forme binaire.

(L'utilité d'un affichage direct en binaire sur un panneau avant a toujours donné lieu à des controverses passionnées dans lesquelles nous n'entrerons pas ici.)

Exercice 1.28 : *Quel est l'avantage de la représentation en complément à deux sur les autres représentations ?*

Exercice 1.29 : *Comment représenteriez-vous 1024 en binaire ? En binaire signé ? En complément à deux ?*

Exercice 1.30 : *Que représente le bit V ? Le programmeur doit-il le tester après une addition ou une soustraction ?*

Exercice 1.31 : *Calculer en complément à deux : + 16, + 17, + 18, − 16, − 17, − 18.*

Exercice 1.32 : *Montrer la représentation hexadécimale du texte suivant, qui a été rangé en format interne ASCII sans parité : « MESSAGE ».*

REPONSES A CERTAINS EXERCICES

1.1 : 254
1.6 : 10000101
1.7 : 11111001
1.8 : 01111111
1.9 : 10000000
1.10 : − 128 et + 127
1.14 : − 32768 et + 32767
1.16 : 00101001 et 10010001
1.17 : illégal. 1010 n'est pas un code DCB correct
1.20 : deux octets (16 bits) en DCB ; 14 bits en complément à deux.

2

ORGANISATION MATERIELLE DU 6502

INTRODUCTION

Pour programmer à un niveau élémentaire, il n'est pas nécessaire de comprendre en détail la structure interne du processeur utilisé. Toutefois, pour programmer de façon efficace, une telle compréhension est nécessaire. Le but de ce chapitre est de présenter les concepts hardware fondamentaux nécessaires pour comprendre le fonctionnement du système 6502. Le système complet comprend non seulement le microprocesseur (ici, le 6502) mais aussi d'autres composants. Ce chapitre présente le 6502 proprement dit. Les autres boîtiers (principalement d'entrées-sorties) seront présentés dans un autre chapitre (Chapitre 7).

Nous allons revoir ici l'architecture de base d'un système à microprocesseur, puis étudier de manière plus détaillée l'organisation interne du 6502. Nous examinerons en particulier les divers registres, puis, le mécanisme de séquencement de l'exécution d'un programme. D'un point de vue hardware, ce chapitre n'offre qu'une présentation simplifiée. Le lecteur intéressé par les détails peut se reporter à notre référence C4 *(« Les Microprocesseurs »).*

ARCHITECTURE DU SYSTEME

L'architecture d'un système microordinateur apparaît Figure 2-1. Le microprocesseur (MPU), qui sera ici un 6502, apparaît sur la gauche de l'illustration. Il remplit les fonctions d'une *unité centrale* (UC) en un seul boîtier : il comprend une *unité arithmétique et logi-*

que (UAL) avec ses registres internes et une *unité de commande* chargée de séquencer le système. Son fonctionnement sera expliqué dans ce chapitre.

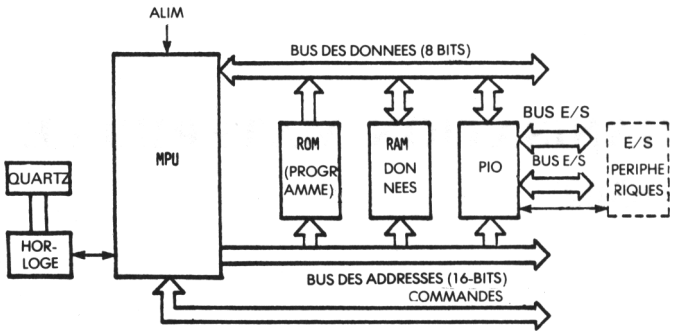


Fig. 2-1 : Architecture d'un système à microprocesseur type

Le MPU crée trois *bus :* un *bus de données* 8-bits bidirectionnel qui apparaît en haut de l'illustration, un *bus d'adresse* 16-bits unidirectionnel et un *bus de commande* qui apparaît en bas de l'illustration. Décrivons la fonction de chacun.

Le *bus de données* transporte les *données* échangées par les différents éléments du système. Généralement, il transfère des données de la mémoire vers le MPU, du MPU vers la mémoire, ou encore du MPU vers un boîtier d'entrées-sorties (les boîtiers d'entrées-sorties sont des composants chargés de communiquer avec les périphériques).

Le *bus d'adresse* transporte une *adresse* générée par le MPU, adresse qui va sélectionner un registre interne à un des boîtiers du système. Cette adresse est un nombre qui spécifie la source ou la destination des données qui transitent par le bus de données.

Le *bus de commande* transporte les différents signaux de synchronisation requis par le système.

Ayant décrit le rôle des bus, connectons maintenant les composants supplémentaires nécessaires à un système complet.

Tout MPU exige une base de temps précise, fournie par une *horloge* ou un *quartz*. Dans la plupart des microprocesseurs plus « anciens », l'oscillateur d'horloge était extérieur au MPU et nécessitait un boîtier supplémentaire. Dans la plupart des microproces-

seurs récents, l'oscillateur d'horloge est généralement incorporé au microprocesseur. Néanmoins, le quartz est toujours externe à cause de sa taille. Le quartz et l'horloge apparaissent à gauche du boîtier MPU sur l'illustration.

Portons maintenant notre attention sur les autres éléments du système. De gauche à droite sur l'illustration nous distinguons :

La ROM qui est la *mémoire à lecture seule* (ou « mémoire morte ») et contient le *programme* du système. L'avantage de la ROM est d'être permanente : son contenu ne disparaît pas quand le système est éteint. Par suite la ROM contient toujours un programme de *chargement initial* ou un *moniteur* (leurs fonctions seront expliquées plus loin) pour permettre le fonctionnement du système au départ. Dans un contexte de contrôle de processus, presque tous les programmes résideront en ROM car ils ne seront probablement jamais changés. Dans ce dernier cas, l'utilisateur industriel doit protéger le système contre les pannes d'alimentation, les programmes ne doivent pas être volatils. Il faut donc qu'ils résident en ROM.

Au contraire, dans un environnement « personnel » (applications individuelles) ou dans un contexte de mise au point de programme (lorsque le programmeur teste son programme). la plupart des programmes résident en RAM pour pouvoir être modifiés facilement. Plus tard, ils pourront résider en RAM ou, si l'on veut, être transférés en ROM. Mais la RAM est volatile : son contenu est perdu lorsqu'on coupe l'alimentation.

La RAM est la *mémoire à lecture et écriture* du système. Dans le cas d'un système de contrôle de processus, la RAM sera généralement petite (elle ne sert qu'aux données). Au contraire, dans un contexte de programme en cours de développement, la quantité de RAM devra être importante, vu qu'elle contiendra les programmes plus le logiciel (« software » = les programmes) d'aide à la mise au point. Toutes les informations doivent être chargées en RAM à partir d'un périphérique, avant usage.

Enfin, le système comporte un ou plusieurs boîtiers d'interface afin de pouvoir communiquer avec le monde extérieur. Le boîtier d'interface le plus fréquemment utilisé est le « *PIO* » (boîtier d'entrées-sorties parallèles). C'est celui qui apparaît sur l'illustration. Le PIO, comme tous les autres boîtiers du système, se connecte aux trois bus et il fournit au moins deux ports de 8-bits pour les communications avec le monde extérieur. Pour plus de détails sur le fonctionnement d'un PIO réel voir notre référence C4 ou encore, pour les détails spécifiques au système 6502, voir le Chapitre 7 (Organes d'entrées-sorties).

Tous ces boîtiers sont connectés aux trois bus, y compris le bus de commande. Toutefois, pour clarifier l'illustration, les connexions entre le bus de commande et ces divers boîtiers ne sont pas détaillées sur le diagramme.

Les modules fonctionnels que nous avons décrits ne résident généralement pas dans un seul boîtier dans le cas de la famille 6502. En fait, nous utiliserons des *boîtiers combinés* qui comprennent à la fois un PIO et une quantité limitée de ROM et de RAM.

Il faut encore d'autres composants pour construire un système réel. En particulier les bus ont généralement besoin d'être *amplifiés*. On peut aussi utiliser une *logique de décodage* pour les mémoires RAM, et enfin certains signaux peuvent nécessiter des « drivers » d'amplification. Ces circuits auxiliaires ne seront pas décrits ici puisqu'ils ne concernent pas la programmation. Le lecteur qui s'intéresse à l'assemblage précis d'un système et aux techniques d'interface pourra se reporter à notre référence C5 (*« Techniques d'Interface aux microprocesseurs »*).

ORGANISATION INTERNE DU 6502

Un diagramme simplifié de l'organisation interne du 6502 apparaît Figure 2-2.

L'unité arithmétique et logique (UAL) apparaît sur la droite de l'illustration. On peut facilement la reconnaître à sa forme en V caractéristique. Le rôle de l'UAL est d'effectuer les opérations arithmétiques et logiques sur les données qui lui sont fournies par ses deux entrées. Les deux entrées de l'UAL sont appelées respectivement « Entrée droite » et « Entrée gauche ». Elles correspondent aux deux extrémités supérieures du « V ». Après avoir effectué une opération arithmétique comme une addition ou une soustraction, l'UAL fournit sa sortie au bas de l'illustration.

L'UAL est liée à un registre spécial, l'*accumulateur* (A). L'accumulateur est sur l'entrée gauche ici. L'UAL se réfère automatiquement à cet accumulateur comme l'une de ses entrées. (Toutefois, il y a moyen de le court-circuiter). C'est une conception classique d'unité centrale d'ordinateur fondée sur l'existence d'un accumulateur. Dans les opérations arithmétiques et logiques, un des opérandes sera l'accumulateur et l'autre sera généralement une case mémoire. Le résultat sera déposé dans l'accumulateur. La référence à l'accumulateur tant comme source que comme destination des données explique son nom : il accumule les résultats.

L'avantage de cette approche qui repose sur l'accumulateur, est la possibilité d'avoir des instructions très courtes ; un seul octet (8 bits)

suffit à spécifier le « code-opération », c'est-à-dire la nature de l'opération à effectuer. Si l'opérande devait être recherché dans l'un des registres (autres que l'accumulateur), il serait nécessaire de consacrer dans l'instruction un certain nombre de bits supplémentaires pour désigner ce registre. Par suite, l'architecture à accumulateur améliore la vitesse d'exécution. L'inconvénient est que l'accumulateur doit toujours être chargé au préalable avec les données désirées. Cela peut aussi entraîner des inefficacités.

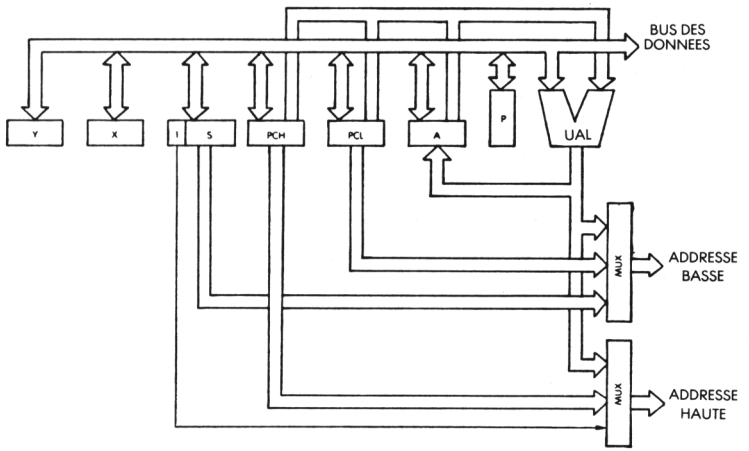


Fig. 2-2 : Organisation interne du 6502

Revenons à l'illustration. A côté de l'UAL, sur la gauche, apparaît un registre 8 bits particulier, le registre des *indicateurs d'état* du processeur (P). Ce registre comporte 8 bits d'état. Chacun de ces bits, réalisé physiquement grâce à une bascule à l'intérieur du registre, est utilisé pour dénoter une condition particulière. Le rôle des différents bits d'état sera expliqué progressivement lors des exemples de programmation qui seront présentés au chapitre suivant et il sera décrit complètement au Chapitre 4 qui présente tout le jeu d'instructions. Par exemple, trois de ces bits d'état sont N. Z et C.

N est l'abréviation de « *Négatif* ». C'est le bit 7, c'est-à-dire le bit le plus à gauche du registre P. Quand ce bit est à un, il indique que le résultat de l'opération que l'UAL vient d'effectuer est négatif.

Le bit Z veut dire « *Zéro* ». Quand ce bit (qui est en position 1) est à 1, cela veut dire qu'on vient d'obtenir un résultat nul.

Le bit C (en position 0, c'est-à-dire le plus à droite), est le bit de « *Retenue* » (carry). Quand on additionne deux nombres de 8 bits et que le résultat ne tient pas en 8 bits, c'est le 9e bit du résultat. La retenue est très utilisée dans les opérations arithmétiques.

Ces bits d'état sont positionnés automatiquement par les différentes instructions. La liste complète des instructions avec la façon dont elles affectent les indicateurs se trouve dans l'appendice A et est également présentée dans le Chapitre 4. Ces bits sont utilisés par le programmeur pour tester diverses conditions spéciales ou exceptionnelles ou encore pour tester rapidement si un résultat est faux. Par exemple, il existe des instructions spéciales pour tester l'indicateur Z : elles permettent de savoir immédiatement si le résultat de l'opération qui précède est nul ou non nul. Toutes les décisions d'un programme en langage assembleur, c'est-à-dire de tous les programmes de ce livre, sont basées sur le test de certains bits. Ces bits seront soit lus à partir du monde extérieur, soit pris parmi les bits d'état de l'UAL. Il est donc très important de comprendre le rôle et l'usage de tous les indicateurs d'état du système. Ici, l'UAL est munie d'un registre qui regroupe ces bits. D'autres boîtiers d'entrées-sorties du système ont eux aussi des bits d'état. Ce sera étudié au Chapitre 7.

Allons maintenant encore plus à gauche sur l'illustration 2-2. Les rectangles horizontaux représentent les registres internes du 6502.

PC est le *compteur ordinal* (program counter). C'est un registre de 16 bits réalisé physiquement sous la forme de deux registres 8 bits : PCL et PCH. PCL (Program Counter Low) est la moitié basse du compteur ordinal, c'est-à-dire les bits 0 à 7. PCH (Program Counter High) est la moitié haute c'est-à-dire les bits 8 à 15. Le compteur ordinal est un registre 16 bits qui contient l'adresse de la prochaine instruction à exécuter. Tout ordinateur possède un compteur ordinal afin de savoir quelle instruction exécuter. Résumons brièvement ici le mécanisme d'accès à la mémoire pour illustrer le rôle du compteur ordinal.

Cycles d'exécution d'une instruction

Reportons-nous maintenant à la Figure 2-3. Le microprocesseur est à gauche et la mémoire apparaît sur la droite. Le boîtier mémoire peut être une ROM, une RAM ou tout autre boîtier qui se trouve renfermer de la mémoire. La mémoire sert à emmagasiner les instructions et les données. Ici, pour montrer le rôle du compteur ordinal, nous allons rechercher une instruction dans la mémoire. Nous supposons que le compteur ordinal a un contenu valable : il contient

une adresse 16 bits qui est l'adresse de la prochaine instruction à chercher en mémoire. Tout processeur procède en trois cycles :

1. Rechercher la prochaine instruction.
2. Décoder l'instruction.
3. Exécuter l'instruction.

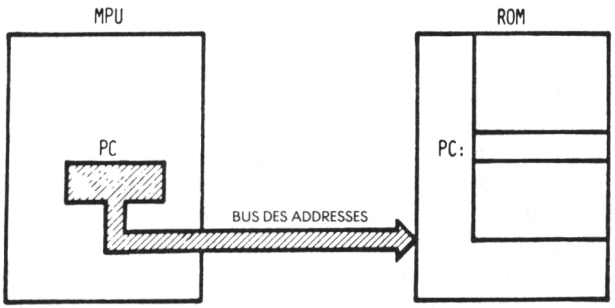


Fig. 2-3 : Recherche d'une instruction en mémoire

Recherche

Suivons la séquence. Lors du premier cycle, le contenu du compteur ordinal est déposé sur le bus adresse et envoyé vers la mémoire (par le bus adresse). Simultanément, un signal « lecture » est activé dans le bus de commande du système (si nécessaire). Cette adresse spécifie un emplacement de la mémoire. Lorsqu'elle reçoit le signal de lecture, la mémoire décode dans ses décodeurs internes l'adresse qu'elle avait reçue et sélectionne la case spécifiée par l'adresse. Quelques centaines de nanosecondes plus tard, la mémoire dépose sur le bus de données les 8 bits de données correspondant à l'adresse spécifiée. Cette donnée 8 bits est l'instruction que nous voulions chercher. Dans notre illustration, cette instruction sera déposée sur le bus de données situé en haut de la figure.

Résumons brièvement la séquence : le contenu du compteur ordinal est sorti sur le bus adresse. Un signal de lecture est généré. La mémoire démarre un cycle. Peut-être 300 nanosecondes après, l'instruction qui se trouvait à l'adresse spécifiée est déposée sur le bus de données. Alors, le microprocesseur lit le bus de données et dépose son contenu dans un registre interne spécial, IR. IR est le registre-instruction. Il a 8 bits de large et il sert à conserver l'instruction qu'on vient de rechercher en mémoire. Le cycle de recherche est

maintenant terminé. Les 8 bits de l'instruction sont maintenant physiquement présents dans le registre interne spécial du 6502, le registre IR. Celui-ci apparaît sur la gauche de la Figure 2-3.

Décodage et exécution

Une fois que l'instruction se trouve dans IR, l'unité de commande du microprocesseur va la décoder et être capable de générer la séquence convenable de signaux internes et externes correspondant à l'instruction concernée. Il y a donc un court délai de décodage suivi par une phase exécution dont la longueur dépend de la nature de l'instruction considérée. Certaines instructions s'exécutent entièrement à l'intérieur du MPU. D'autres recherchent ou envoient des données en mémoire. C'est pourquoi les différentes instructions du 6502 ont des durées d'exécution différentes. Cette durée est exprimée en cycles d'horloge. L'appendice donne le nombre de cycles demandé par chaque instruction. Le 6502 normal utilise une horloge à 1 MHz. La durée de chaque cycle est alors 1 microseconde. Mais comme différentes versions peuvent utiliser différentes fréquences d'horloge, la vitesse d'exécution s'exprime en nombre de cycles plutôt qu'en nombre de microsecondes.

On remarque aussi, tout à fait à gauche de l'illustration, que le 6502 possède un oscillateur interne. C'est l'*horloge,* qui dans le cas du 6502, est interne au microprocesseur.

Recherche de la prochaine instruction

Nous avons décrit comment, à l'aide du compteur ordinal, on peut rechercher une instruction en mémoire. Lors de l'exécution d'un programme, les instructions sont recherchées en mémoire *à la suite.* Il faut donc un mécanisme automatique pour rechercher les instructions *en séquence.* Cette tâche est assurée par un simple incrémenteur lié au compteur ordinal. Ceci est illustré Figure 2-4. A chaque fois que le contenu du compteur ordinal (en bas de l'illustration) est placé sur le bus-adresse, il est incrémenté puis réécrit dans le compteur ordinal. Par exemple, si le compteur ordinal contenait 0, on sortirait 0 sur le bus adresse. Alors le contenu du compteur ordinal serait incrémenté et la valeur 1 serait écrite dans le compteur ordinal. De cette manière, la prochaine fois qu'on utilisera le compteur ordinal, c'est l'instruction située à l'adresse 1 qui sera obtenue. Nous venons de réaliser un mécanisme automatique pour séquencer les instructions.

Il faut insister sur le fait que les descriptions ci-dessus sont simplifiées. En réalité, certaines instructions font 2 ou même 3 octets de long de sorte qu'il faut de cette manière rechercher en mémoire des

octets successifs. Néanmoins, le mécanisme est identique. Le compteur ordinal est utilisé pour rechercher les octets successifs d'une instruction aussi bien que pour rechercher les instructions successives elles-mêmes. Le compteur ordinal, avec son incrémenteur, fournit un mécanisme automatique pour pointer vers des emplacements mémoire successifs.

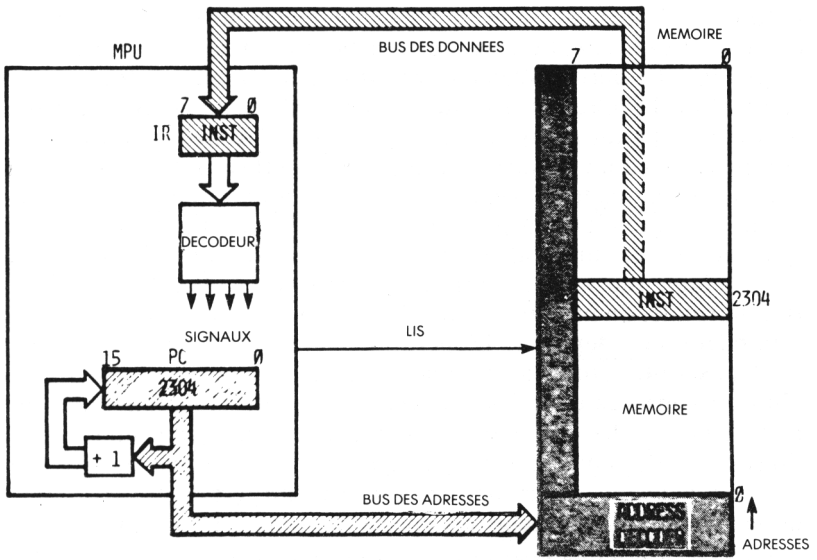


Fig. 2-4 : Séquencement automatique

Autres registres du 6502

Une dernière partie de la Figure 2-2 n'a pas encore été expliquée. C'est l'ensemble des trois registres appelés X, Y et S. Les registres X et Y sont appelés registres index. Ils ont 8 bits de large. Ils peuvent servir à renfermer des données sur lesquelles le programme travaille. Toutefois, leur usage normal est celui de registres index.

Le rôle des registres index sera décrit dans le Chapitre 5 sur les Techniques d'adressage. Disons brièvement que le contenu de ces deux registres index peut être ajouté de plusieurs façons à une adresse spécifiée dans le système, fournissant ainsi un déplacement automatique. C'est une facilité importante pour retrouver efficacement des données lorsqu'elles sont rangées dans des tables. Ces deux registres ne sont pas tout à fait équivalents et leurs rôles différents seront explicités dans le chapitre sur les Techniques d'adressage.

Le registre S a pour rôle de contenir un pointeur vers le sommet de la zone mémoire affectée à la pile.

Introduisons maintenant le concept formel de pile.

La pile

Une pile forme une structure LIFO (« last in, first out » = dernier entré, premier sorti). Une pile est un ensemble de registres ou d'emplacements mémoire, alloués à une telle structure de données. La caractéristique essentielle de cette structure est son caractère chronologique. Le dernier élément introduit dans la pile est toujours au sommet de la pile. On peut développer une analogie avec une pile d'assiettes sur un comptoir de restaurant. Il y a un trou dans le comptoir, avec un ressort au fond. Les assiettes sont empilées dans le trou. Avec une telle organisation, il est garanti que l'assiette qui a été mise la première (la plus ancienne), est toujours au fond. Celle qui a été placée sur la pile le plus récemment, se trouve sur le dessus. Cet exemple illustre une autre caractéristique de la pile. En utilisation normale, on ne peut accéder à la pile que par deux instructions : « empiler » et « dépiler ». L'opération « empiler » (« push ») permet de déposer un élément au sommet de la pile. En pratique, dans le cas d'un microprocesseur, c'est *l'accumulateur* qui sera déposé au sommet de la pile. Le « dépilage » (« pull ») se ramènera à un transfert du sommet de la pile dans l'accumulateur. D'autres instructions spécialisées peuvent exister permettant des transferts entre le sommet de la pile et d'autres registres spéciaux comme le registre d'état.

La présence d'une pile est nécessaire pour permettre trois facilités de programmation dans le système : les sous-programmes, les interruptions et le stockage de données temporaires. Le rôle de la pile lors des sous-programmes sera expliqué au Chapitre 3 (Techniques de base de la programmation). Le rôle de la pile lors des interruptions sera expliqué au Chapitre 6 (Techniques d'entrées-sorties). Enfin, le rôle de la pile pour sauvegarder des données à grande vitesse sera expliqué avec des programmes d'application spécifiques.

Nous supposerons simplement maintenant que la pile est une nécessité dans tout ordinateur. Une pile peut être réalisée de deux manières.

1. Un certain nombre fixé de registres peut lui être consacré dans le microprocesseur lui-même. C'est une « pile hardware ». Elle a l'avantage de la vitesse. Mais elle a l'inconvénient du nombre limité de registres.

2. La plupart des microprocesseurs d'usage général choisissent une autre approche, la « pile software » afin de ne pas restreindre la pile à un trop petit nombre de registres. Telle est l'approche choisie

par le 6502. Dans l'approche software, un registre spécial du microprocesseur, ici le registre S, renferme le pointeur de pile, c'est-à-dire l'adresse du sommet de la pile (ou, plus précisément, cette adresse + 1). La pile est donc implantée dans une zone mémoire. Le pointeur de pile aura donc besoin de 16 bits pour pouvoir pointer n'importe où en mémoire.

Cependant, dans le cas du 6502, le pointeur de pile est restreint à 8 bits. Il a un 9e bit à gauche, toujours à 1. En d'autres termes, la zone consacrée à la pile dans le cas du 6502, va des adresses 256 à 511. En binaire, cela s'écrit « 100000000 » à 111111111 ». La pile commence toujours à l'adresse la plus haute et elle progresse « en arrière » : le pointeur de pile est décrémenté lorsqu'on empile.

Pour utiliser la pile, le programmeur n'a qu'à initialiser le registre S. Tout le reste est automatique.

On dit que la pile réside dans la *page 1* de la mémoire. Introduisons maintenant le concept de *pagination.*

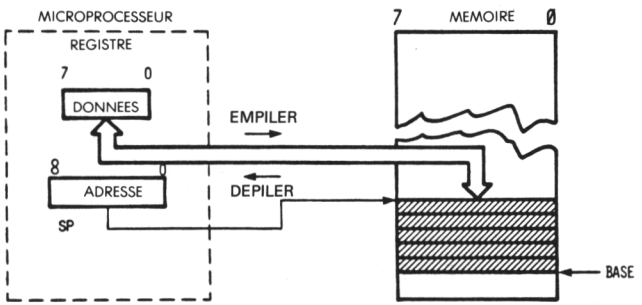


Fig. 2-5 : Les deux instructions de manipulation de la pile

Le concept de la pagination

Le microprocesseur 6502 possède un bus adresse de 16 bits. 16 bits peuvent former jusqu'à $2^{16} = 64$ K combinaisons (1K = 1024). En raison des caractéristiques d'adressage du 6502 qui seront présentées au Chapitre 5, il est commode de partitionner la mémoire en *pages.* Une page n'est autre qu'un bloc de 256 mots. Ainsi, les emplacements 0 à 255 forment la page 0 de la mémoire. Elle servira pour le mode d'adressage « page zéro ». La page 1 de la mémoire comprend des cases 256 à 511. Nous venons de voir que la page 1 est normalement réservée pour la pile. Toutes les autres pages sont considérées comme libres et peuvent être utilisées comme on veut.

Dans le cas du 6502, il est important de garder à l'esprit l'organisation paginée de la mémoire. Lorsqu'on a à franchir une frontière de page, cela peut souvent introduire un délai supplémentaire d'un cycle pour l'exécution.

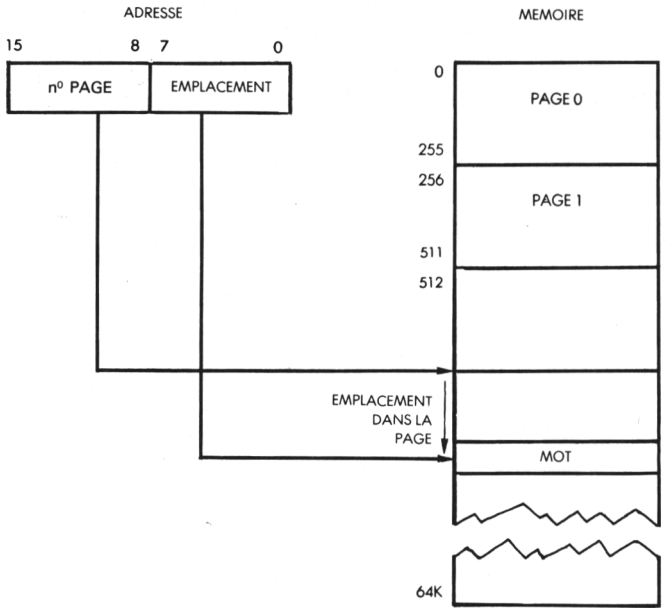


Fig. 2-6 : Concept de page

LE BOITIER 6502

Pour finir notre description du schéma de la Figure 2-2, le bus de données qui figure en haut représente le bus de données externe. Il servira à communiquer avec les boîtiers extérieurs, en particulier la mémoire. AO-7 et A8-15 représentent respectivement la partie basse et la partie haute du bus adresse créé par le 6502.

Pour être complet, nous présentons ici le brochage exact du microprocesseur 6502. Il est inutile de lire cette section en détail pour comprendre le reste du livre. Toutefois, si vous souhaitez connecter des boîtiers pour former un système, cette description est nécessaire.

Le brochage du 6502 apparaît Figure 2-7. Le bus de données est appelé DB0-7 et il se reconnaît facilement sur la droite de la figure. Le bus adresse est marqué A0-11 et A12-15. Il va des broches 9 à 20 à gauche du boîtier et 22 à 25 sur la droite.

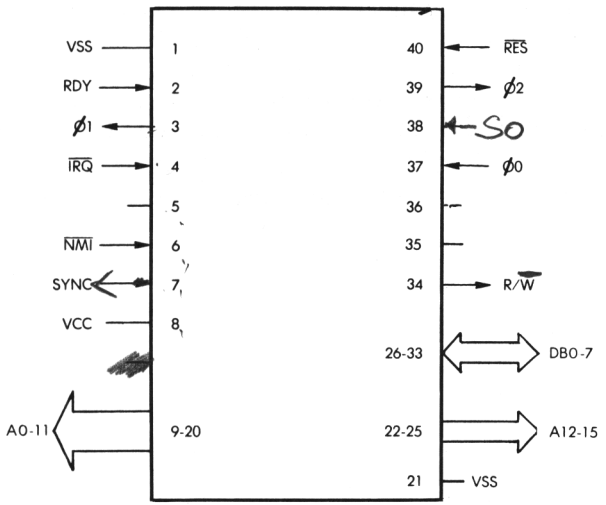


Fig. 2-7 : Brochage du 6502

Les autres signaux sont les signaux de commande et l'alimentation.

Les signaux de commande

— R/$\overline{W}$: est la ligne lecture-écriture qui gouverne la direction des transferts de données sur le bus de données.
— IRQ et NMI sont « demande d'interruption » et « interruption non masquable ». Ce sont les deux lignes de demande d'interruption qui seront utilisées au Chapitre 7.
— SYNC est un signal qui indique au monde extérieur qu'on est en train de rechercher le code opération d'une instruction.
— RDY est généralement utilisé pour se synchroniser sur une mémoire lente : il arrête le processeur.
— SO met à un l'indicateur de débordement. Il est rarement utilisé.
— $Ø_0$, $Ø_1$ et $Ø_2$ sont les signaux d'horloge.
— RES est la remise à zéro (RESET) qui permet l'initialisation.
— Vss et Vcc sont les broches d'alimentation (5V).

RECAPITULATION HARDWARE

Ceci termine notre description hardware de l'organisation interne du 6502. La structure exacte des bus internes du 6502 n'a pas d'importance ici. Toutefois, le rôle exact de chaque registre est fondamental, et le lecteur devrait l'avoir parfaitement compris avant de continuer. Si vous êtes familier avec des concepts qui viennent d'être présentés, lisez la suite. Si vous ne vous sentez pas sûr de certains d'entre eux, il vous est suggéré de relire les sections appropriées de chapitre car ces connaissances seront nécessaires dans la suite. Il vous est aussi suggéré de revoir la Figure 2-2 et de vous assurer que vous comprenez la fonction de chaque registre de l'illustration.

Examinons maintenant la programmation et ses règles de base.

3

TECHNIQUES DE BASE DE LA PROGRAMMATION

INTRODUCTION

Le but de ce chapitre est de présenter toutes les techniques de base nécessaires pour écrire un programme sur le 6502. Ce chapitre introduira des notions complémentaires telles que la gestion des registres, les boucles et les sous-programmes. Il mettra l'accent sur des techniques de programmation utilisant seulement les ressources *internes* du 6502, c'est-à-dire les registres. Des programmes pratiques seront développés, par exemple arithmétiques. Ces programmes serviront à illustrer les différents concepts présentés et ils feront appel à de véritables instructions. Ainsi, on verra comment on peut utiliser certaines instructions pour manipuler des informations entre la mémoire et le MPU et d'autres à l'intérieur du MPU proprement dit. Le chapitre suivant discutera alors en détail les instructions disponibles sur le 6502. Le Chapitre VI présentera les techniques dont on dispose pour manipuler les informations *en dehors* du 6502 c'est-à-dire les techniques d'entrées-sorties.

Dans ce chapitre, nous allons essentiellement apprendre « par la pratique ». En examinant des programmes de complexité croissante, nous allons apprendre le rôle des différentes instructions et des registres, et nous appliquerons les concepts développés jusque-là. Néanmoins, une notion importante ne sera pas présentée ici : celle des techniques d'adressage. Vu son apparente complexité, elle sera présentée à part, dans le Chapitre V.

Commençons immédiatement à écrire des programmes pour le 6502. Nous commencerons par des programmes arithmétiques.

PROGRAMMES ARITHMETIQUES

Les programmes arithmétiques sont essentiellement l'addition et la soustraction, plus multiplication et division. Les programmes présentés ici opèreront sur des entiers. Ces entiers peuvent être des nombres binaires positifs, ou alors, s'ils ont un signe, ils seront exprimés en complément à deux, auquel cas le bit le plus à gauche est le bit de signe (voir le Chapitre 1 pour un rappel sur la notation en complément à 2).

Addition sur 8 bits

Nous allons ajouter deux opérandes de 8 bits appelés OP1 et OP2, rangés respectivement aux adresses ADR1 et ADR2 de la mémoire. La somme sera appelée RES et elle occupera l'adresse mémoire ADR3. Ceci est illustré Figure 3-1. Le programme qui effectue cette addition est le suivant :

```
LDA    ADR1    CHARGER OP1 DANS A
ADC    ADR2    AJOUTER OP2 A OP1
STA    ADR3    SAUVER RES EN ADR3
```

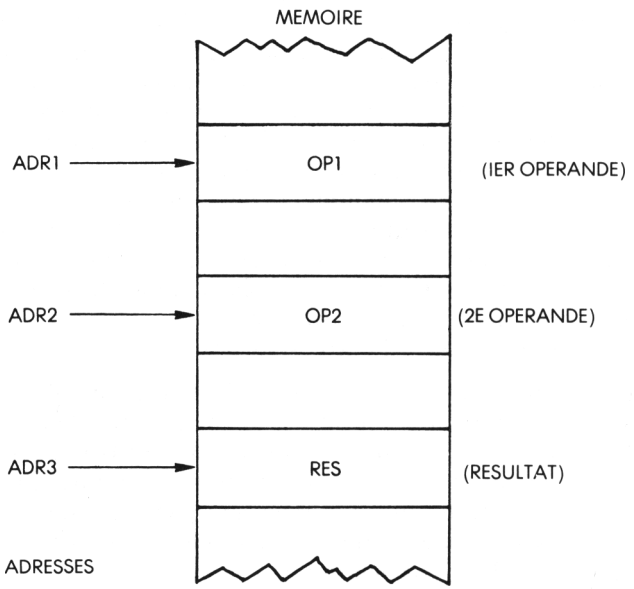


Fig. 3-1 : Addition sur 8 bits. RES = OP1 + OP2

Le programme nécessite trois instructions. Chaque ligne est une *instruction symbolique.* Chacune de ces instructions sera traduite par l'assembleur en 1, 2 ou 3 octets binaires. Nous ne nous préoccuperons pas ici de la traduction et nous ne considérerons que la représentation symbolique. La première ligne est une instruction « LDA ». LDA veut dire « charger l'accumulateur A avec le contenu de l'adresse qui suit ».

L'adresse spécifiée sur la première ligne est ADR1. ADR1 est la représentation symbolique d'une adresse binaire de 16 bits. Le symbole ADR1 sera défini ailleurs dans le programme. Ce pourrait être, par exemple, l'adresse « 100 » (en décimal).

L'instruction LDA ordonne de « charger l'accumulateur A » (à l'intérieur du 6502) à partir de la case mémoire d'adresse 100. Il en résultera une opération de lecture à l'adresse 100, dont le contenu sera transmis le long du bus de données, et déposé dans l'accumulateur. Rappelons que les opérations arithmétiques et logiques considèrent l'accumulateur comme l'un des opérandes de départ (cf. le chapitre précédent pour plus de détails). Comme nous voulons additionner les deux valeurs OP1 et OP2, nous devons d'abord charger OP1 dans l'accumulateur. Nous pourrons ensuite ajouter le contenu de l'accumulateur (OP1) à OP2.

Le champ à droite de l'instruction s'appelle la *zone commentaire.* Il est ignoré par le processeur, mais il sert à faciliter la lecture du programme. Pour comprendre ce que fait un programme, il est d'une importance fondamentale de disposer de bons commentaires à l'intérieur de celui-ci. Ceci s'appelle *documenter* un programme. Ici, le commentaire est évident. La valeur OP1, qui se trouve à l'adresse ADR1 est chargée dans l'accumulateur A.

Le résultat de cette première instruction est illustré par la Figure 3-2.

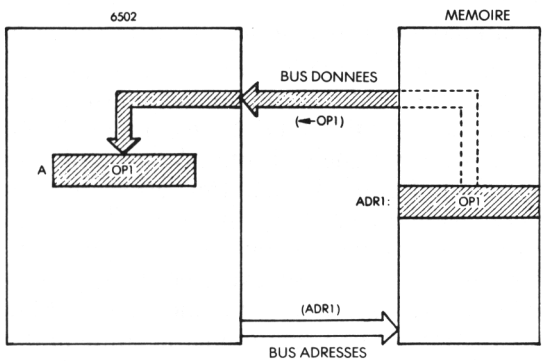


Fig. 3-2 : LDA ADR1 : OP1 est chargé depuis la mémoire

La seconde instruction de notre programme est : ADC ADR2. Elle ordonne d'« ajouter le contenu de l'adresse-mémoire ADR2 à l'accumulateur ». En se reportant à la Figure 3-1, on voit que le contenu de l'adresse mémoire ADR2 est OP2, notre second opérande. Le contenu de l'accumulateur est actuellement OP1, notre premier opérande. L'effet de l'exécution de la seconde instruction sera que OP2 soit recherché en mémoire et ajouté à OP1. La somme sera déposée dans l'accumulateur. Le lecteur se rappellera que le résultat d'une opération arithmétique est, dans le cas du 6502, déposé dans l'accumulateur. Avec d'autres processeurs, il peut être possible de déposer ce résultat dans d'autres registres ou en mémoire.

La somme de OP1 et OP2 est maintenant dans l'accumulateur. Nous n'avons plus qu'à transférer le contenu de l'accumulateur dans la case-mémoire ADR3 pour ranger le résultat à l'emplacement demandé. Là encore, le champ le plus à droite de l'instruction est un commentaire qui décrit l'effet de l'instruction (ajouter OP2 à A).

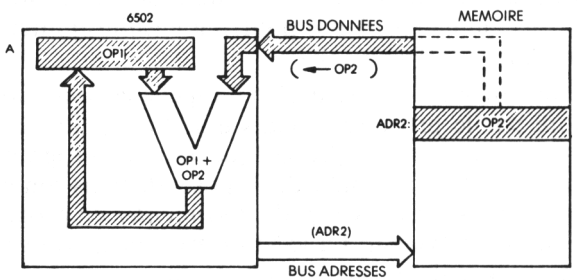


Fig. 3-3 : ADC ADR2

L'effet de la seconde instruction est illustré sur la Figure 3-3.

On peut vérifier sur la Figure 3-3 que, au départ, l'accumulateur contenait OP1. Après l'addition, un nouveau résultat a été écrit dans l'accumulateur, à savoir OP1 + OP2. Le contenu de n'importe quel registre du système, ainsi que de toute case mémoire, reste inchangé lorsqu'on opère une lecture. En d'autres termes, *lire un registre ou un emplacement mémoire ne change pas son contenu.* C'est une opération d'écriture, et exclusivement celle-ci, qui peut modifier le contenu d'un registre. Dans cet exemple, les contenus des cases-mémoire ADR1 et ADR2 restent inchangés. Mais, après la deuxième instruction de ce programme, le contenu de l'accumulateur est modifié puisqu'on y écrit la sortie de l'UAL. Son contenu précédent est alors perdu.

Préservons ce résultat à l'adresse ADR3 et nous aurons terminé notre addition élémentaire.

La troisième instruction est STA ADR3. Cela signifie « ranger le contenu de l'accumulateur A à l'adresse ADR3 ». Son effet est clair et est illustré sur la Figure 3-4.

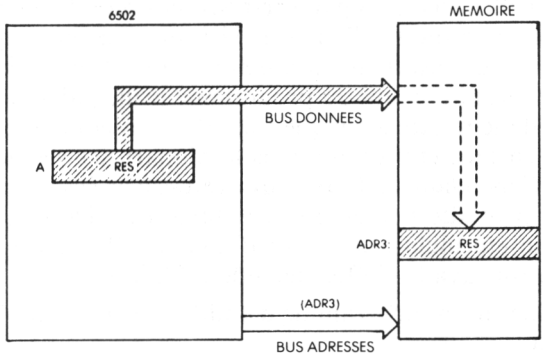


Fig. 3-4 : STA ADR3 (ranger l'accumulateur en mémoire)

Particularités du 6502

Le programme de trois instructions ci-dessus constituerait effectivement le programme complet pour la plupart des microprocesseurs. Mais le 6502 a deux particularités qui vont nécessiter, dans le cas général, deux instructions supplémentaires.

D'abord, l'instruction ADC signifie en fait « *ajouter avec la retenue* » (« Add with Carry ») et non « ajouter ». La différence est qu'une instruction d'addition simple ajoute deux nombres. Une addition avec retenue ajoute les deux nombres plus la valeur du bit de retenue. Comme ici, nous additionnons des nombres de 8 bits, aucune retenue ne devrait intervenir. Etant donné qu'au moment où nous commençons l'addition nous ne connaissons pas forcément l'état du bit de retenue (il peut avoir été mis à un par une instruction précédente), nous devons l'annuler, c'est-à-dire le mettre à zéro. Cela sera accompli par l'instruction CLC (clear carry = annuler la retenue).

Malheureusement, le 6502 n'a pas les deux types d'instructions d'addition (avec et sans retenue). Il n'a que ADC. Il en résulte que pour des additions sur un seul octet, il est nécessaire de toujours prendre la précaution de mettre à zéro le bit de retenue. Il ne s'agit pas d'un inconvénient majeur mais ceci ne doit pas être négligé.

La seconde particularité du 6502 réside dans le fait qu'il possède des instructions *décimales* que nous utiliserons dans la section suivante sur l'arithmétique DCB. Le 6502 fonctionne sous l'un ou l'autre de deux modes mutuellement exclusifs : binaire ou décimal. L'état dans lequel il se trouve est déterminé par un bit d'état, le bit « D » (du registre P). Comme nous voulons opérer un mode binaire dans cet exemple, il faut s'assurer que le bit D est correctement initialisé. Ceci sera accompli par une instruction CLD (clear D) qui va mettre à zéro le bit D. Bien sûr, si toute l'arithmétique du programme doit être effectuée en binaire, le bit D sera mis à zéro une fois pour toutes au début du programme principal, et il ne sera pas nécessaire de le refaire à chaque fois. Par conséquent, cette instruction peut, en fait, être omise dans la plupart des segments de programmes. Cependant, le lecteur qui va essayer ces exercices sur un ordinateur peut alterner les exercices binaires et DCB : cette instruction a donc été incluse ici étant donné qu'elle doit apparaître au moins une fois avant d'effectuer une addition binaire.

En résumé, notre programme d'addition 8 bits complet et sûr est maintenant :

```
CLC             ANNULER LE BIT DE RETENUE
CLD             ANNULER LE BIT DE MODE DECIMAL
LDA   ADR1      CHARGER OP1 EN A
ADC   ADR2      AJOUTER OP2 A OP1
STA   ADR3      RANGER RES EN ADR3
```

On peut utiliser des adresses physiques effectives au lieu de ADR1, ADR2, ADR3. Si on désire garder des adresses symboliques, il sera nécessaire d'utiliser des « pseudo-instructions » qui spécifient la valeur de ces adresses symboliques de sorte que l'assembleur puisse, lors de sa traduction, substituer les adresses physiques effectives. De telles pseudo-instructions seraient, par exemple :

```
ADR1 = $100
ADR2 = $120
ADR3 = $200
```

Exercice 3.1 : *Fermez ce livre (consultez seulement la liste des instructions à la fin du livre). Ecrivez un programme qui additionne deux nombres rangés aux emplacements LOC1 et LOC2 de la mémoire. Déposez le résultat dans la mémoire LOC3. Comparez alors votre programme avec le programme ci-dessus.*

Addition sur 16 bits

L'addition 8 bits n'autorise que l'addition de nombres de 8 bits, c'est-à-dire de nombres compris entre 0 et 255 si l'on est en binaire

absolu. Dans la plupart des applications pratiques, il est nécessaire de faire appel à la *multi-précision* et d'ajouter des nombres de 16 bits ou plus. Nous présenterons ici des exemples d'arithmétique sur 16 bits. Ils peuvent être facilement étendus à 24 ou 32 bits ou même plus (on utilise toujours des multiples de 8 bits). Nous supposerons que le premier opérande est stocké aux adresses-mémoire ADR1 et ADR1-1. Comme OP1 est maintenant un nombre de 16 bits, il a besoin de deux emplacements-mémoire de 8 bits. De même, OP2 sera rangé en ADR2 et ADR2-1. Le résultat sera déposé aux adresses-mémoire ADR3 et ADR3-1. C'est illustré sur la Figure 3-5.

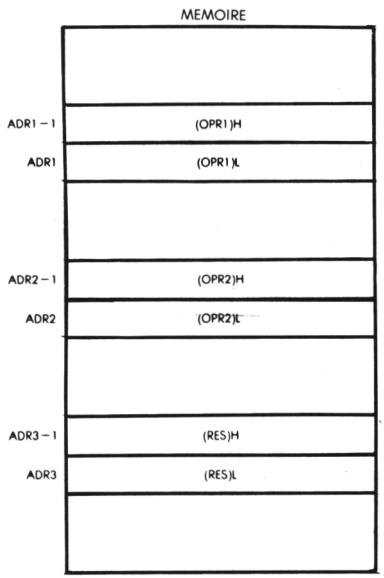


Fig. 3-5 : Addition 16 bits : les opérandes

La logique de ce programme est tout à fait semblable au précédent. On ajoute d'abord la moitié inférieure des deux opérandes puisque le microprocesseur ne peut ajouter que 8 bits à la fois. La retenue engendrée par l'addition de ces deux octets de poids faible sera automatiquement mémorisée dans le bit de retenue (« C ») à l'intérieur du microprocesseur. Ensuite, on ajoute les deux octets de poids fort avec la retenue quelle qu'elle soit et on préserve le résultat en mémoire. Le programme apparaît ci-dessous.

```
CLC
CLD
LDA    ADR1      PARTIE BASSE DE OP1
ADC    ADR2      (OP1 + OP2) PARTIE BASSE
STA    ADR3      RANGER LA PARTIE BASSE DE RES
LDA    ADR1-1    PARTIE HAUTE DE OP1
ADC    ADR2-1    (OP1 + OP2) HAUT + RETENUE
STA    ADR3-1    RANGER LA PARTIE HAUTE DE RES
```

Les deux premières instructions du programme (CLC et CLD) sont là à titre de précautions. Leur rôle a été expliqué dans la section précédente. Examinons la suite. Les trois instructions suivantes sont essentiellement identiques à celles de l'addition 8 bits du paragraphe précédent. Elles ont pour effet d'ajouter les moitiés de faible poids (bit 0 à 7) de OP1 et OP2. La somme appelée RES est rangée à l'adresse mémoire ADR3.

Dès qu'une addition est effectuée, toute retenue qui en résulte est automatiquement rangée dans le bit de retenue (« carry ») du registre des indicateurs (registre P). Si les deux nombres de 8 bits ne génèrent pas de retenue, la valeur du bit de retenue sera zéro. Si les deux nombres engendrent une retenue, alors le bit C vaudra 1.

Les trois instructions suivantes sont elles-aussi identiques au programme d'addition 8 bits qui précède. Elles ajoutent les parties de poids fort (bit 8 à 15) de OP1 et OP2, plus la retenue, quelle qu'elle soit, et rangent le résultat à l'adresse ADR3-1. Lorsque le programme est exécuté, le résultat 16 bits est rangé aux adresses-mémoire ADR3 et ADR3-1.

On suppose ici qu'aucune retenue n'est produite par cette addition 16 bits. On suppose donc que le résultat est bien un nombre de 16 bits. Si le programmeur soupçonne pour une raison ou pour une autre que le résultat risque d'avoir 17 bits, alors il faudrait ajouter des instructions qui testent le bit de retenue après l'addition. L'emplacement des opérandes en mémoire est illustré sur la Figure 3-5.

Note : Nous avons supposé ici que la partie haute des opérandes est rangée *au-dessus* de la partie basse, c'est-à-dire à l'adresse-mémoire inférieure. Cela n'est pas nécessairement le cas. En fait, dans le 6502, les adresses sont rangées de façon inverse : la partie basse est rangée la première en mémoire et la partie haute est rangée à l'adresse suivante. Pour utiliser la même convention pour les adresses et les données, il est recommandé que les données soient elles aussi stockées la partie basse avant la partie haute. Ceci est illustré sur la Figure 3-6.

Exercice 3.2 : *Réécrire le programme d'addition 16-bits ci-dessus avec l'implantation mémoire de la Figure 3-6.*

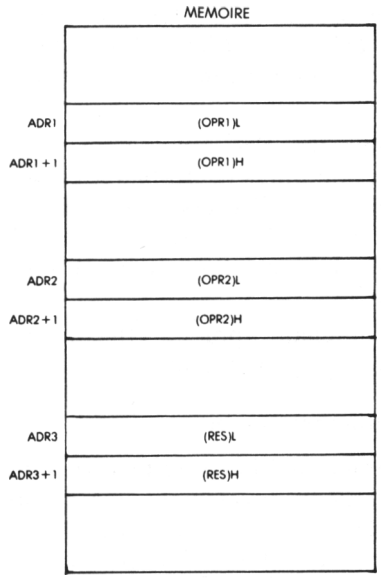


Fig. 3-6 : Rangement des opérandes dans l'ordre inverse

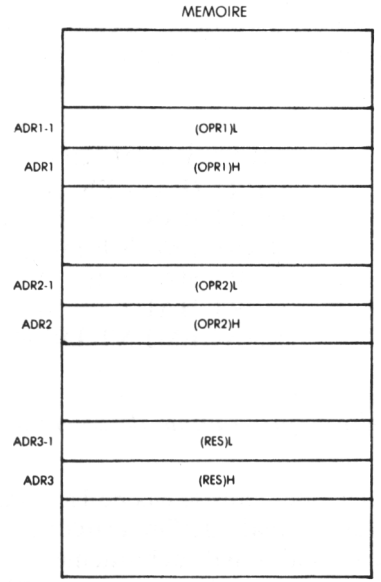


Fig. 3-6.1 : Cas où l'on pointe vers l'octet haut

Exercice 3.3 : *Supposons maintenant que ADR1 ne pointe pas vers la partie basse de OPR1 (cf. Figure 3-6) mais vers la partie haute : ceci est illustré sur la Figure 3-6.1. Ecrire à nouveau le programme correspondant.*

C'est le programmeur, c'est-à-dire *vous,* qui devez décider de la façon de ranger les nombres de 16-bits (c'est-à-dire partie haute ou basse d'abord) et aussi si l'adresse pointe vers l'octet haut ou bas. C'est le premier, parmi beaucoup d'autres, des choix que vous apprendrez à faire lorsqu'il faut élaborer un algorithme ou une structure de données.

Nous avons maintenant appris à effectuer une addition binaire. Tournons nous vers la soustraction.

Soustraction de nombres 16 bits

Faire une soustraction 8 bits serait trop simple. Laissons-le à titre d'exercice et effectuons directement une soustraction sur 16 bits. Comme d'habitude, nos deux nombres OPR1 et OPR2 sont rangés aux adresses ADR1 et ADR2. L'implantation mémoire sera, par exemple, celle de la Figure 3-6. Pour soustraire, l'instruction de soustraction (SBC) sera utilisée au lieu de celle d'addition (ADC). Le seul changement supplémentaire par rapport à l'addition est que nous utiliserons un SEC (Set Carry) au début du programme, au lieu d'un CLC. SEC veut dire « mettre le bit de retenue à 1 ». Cela constitue une condition d'« absence d'emprunt ». Le reste du programme est identique à celui de l'addition. Le programme apparaît ci-dessous :

```
CLD
SEC                    MISE A 1 DU BIT DE RETENUE
LDA     ADR1           OPR1 (BAS) EN A
SBC     ADR2           OPR1 BAS — OPR2 BAS
STA     ADR3           RANGEMENT DE RESULTAT BAS
LDA     ADR1 + 1       OPR1 HAUT EN A
SBC     ADR2 + 1       OPR1 HAUT — OPR2 HAUT
STA     ADR3 + 1       RANGEMENT DE RESULTAT HAUT
```

Exercice 3.4 : *Ecrire le programme de soustraction pour des opérandes de 8 bits.*

Il faut se rappeler que, dans le cas de l'arithmétique en complément à deux, la valeur finale de l'indicateur de retenue n'a pas de signification. Si une condition de débordement s'est produite lors de la soustraction, alors le bit de débordement (bit V) du registre d'état est positionné. On peut alors le tester.

Les exemples qu'on vient de présenter concernent l'addition binaire simple. Cependant un autre type d'addition peut être nécessaire : l'addition DCB.

Airthmétique DCB

Addition DCB sur 8 bits

La notion d'arithmétique DCB a été présentée au Chapitre 1. Elle est utilisée essentiellement dans des applications de gestion où il est impératif de conserver tous les chiffres significatifs dans les résultats. Dans la notation DCB, on utilise un quartet (4 bits) pour représenter un chiffre décimal (0 à 9). Il en résulte qu'un octet peut contenir deux chiffres DCB (cela s'appelle DCB *compacté).* Ajoutons maintenant deux octets qui contiennent chacun deux chiffres DCB.

Pour reconnaître les problèmes, essayons d'abord quelques exemples numériques :

Ajoutons d'abord « 01 » et « 02 » :

« 01 » a pour représentation 00000001
« 02 » a pour représentation 00000010

Le résultat est 00000011

C'est la représentation DCB de « 03 ». (Si vous n'en êtes pas sûr, regardez la table de conversion à la fin du livre). Tout a bien marché dans ce cas. Essayons un autre exemple.

« 08 » a pour représentation 00001000
« 03 » a pour représentation 00000011

Exercice 3.5 : *Calculer la somme des deux nombres ci-dessus en représentation DCB. Qu'obtenez-vous ? (la réponse est donnée ci-dessous).*

Si vous obtenez 00001011, c'est que vous avez calculé la somme de « 8 » et « 3 » en *binaire.* Vous avez obtenu « 11 » en *binaire.* Malheureusement, « 1011 » est illégal en DCB. Ce qu'il faudrait, c'est obtenir la représentation DCB de « 11 » soit « 00010001 » !

Le problème provient du fait que la notation DCB n'utilise que les dix premières combinaisons de 4 chiffres pour représenter les symboles décimaux « 0 » à « 9 ». Les six autres combinaisons possibles sont inutilisées et la combinaison illégale « 1011 » est l'une de celles-ci. En d'autres termes, chaque fois que la somme de deux chiffres est supérieure à « 9 », il faut ajouter « 6 » au résultat afin de « sauter » les

six combinaisons inutilisées. Ajoutons la représentation binaire de « 6 » à « 1011 » :

```
      1011   (résultat illégal)
+     0110   (+ 6)
le résultat est....10001
```

C'est bien « 11 » en représentation DCB. Nous avons maintenant obtenu le résultat correct. Il faut donc compenser pour les six codes manquants. Sur la plupart des microprocesseurs, on doit utiliser une instruction spéciale appelée « ajustement décimal » (decimal adjust) pour ajuster le résultat de l'addition binaire (ajouter 6 si le résultat est supérieur à 9). Dans le cas du 6502, l'instruction ADC le fait automatiquement. C'est un avantage du 6502 pour faire de l'arithmétique DCB.

Le problème suivant est apparent sur le même exemple. Dans notre exemple, une retenue est générée entre le chiffre DCB le moins significatif (le plus à droite) et le chiffre de gauche. Cette retenue interne doit être prise en compte et ajoutée au second chiffre DCB. L'instruction d'addition du 6502 en tient compte automatiquement. Néanmoins, il est souvent commode de détecter cette retenue interne du bit 3 vers le bit 4 (« retenue intermédiaire »). Le 6502 n'a pas d'indicateur correspondant.

Finalement, tout comme dans le cas de l'addition binaire, les instructions habituelles SED et CLC doivent être placées avant l'addition proprement dite. Par exemple, un programme qui additionne les nombres « 11 » et « 22 » apparaît ci-dessous :

```
CLC          ANNULER RETENUE
SED          METTRE EN MODE DECIMAL
LDA #$11     LITTERAL DCB « 11 »
ADC #$22     LITTERAL DCB « 22 »
STA ADR
```

Nous utilisons deux nouveaux symboles dans ce programme : « # » et « $ ». Le symbole « # » avertit qu'il est suivi par un « littéral » (ou une constante). Le signe « $ » dans la zone opérande de l'instruction indique que la donnée qui le suit est exprimée en hexadécimal.

Les représentations hexadécimales et DCB des chiffres « 0 » à « 9 » sont identiques. Ici nous voulons additionner les littéraux (ou constantes) « 11 » et « 22 ». Le résultat est rangé à l'adresse ADR. Quand l'opérande est spécifié comme partie de l'instruction comme c'est le cas dans l'exemple ci-dessus, on dit qu'il s'agit d'*adressage immédiat* (les différents modes d'adressage seront étudiés en détail au

Chapitre V). Le rangement du résultat à une adresse spécifiée comme STA ADR s'appelle de l'*adressage absolu,* lorsque ADR représente une adresse, normalement sur 16 bits.

Exercice 3.6 : *Pourrions-nous placer l'instruction CLC du programme en dessous de l'instruction LDA ?*

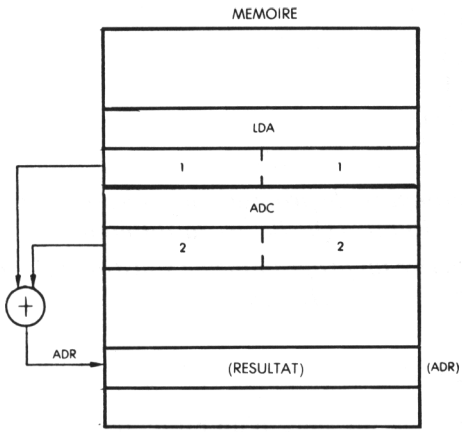


Fig. 3-7 : Stockage de chiffres DCB

Soustraction DCB

La soustraction DCB est complexe en apparence. Pour effectuer une soustraction DCB, il faut ajouter le *complément à 10* du nombre, exactement de la même façon qu'on ajoute le complément à 2 pour faire une soustraction binaire. Le complément à 10 s'obtient en calculant le complément à 9 puis en ajoutant 1. Ceci demande généralement trois à quatre opérations sur un microprocesseur standard. Néanmoins, le 6502 possède une instruction spéciale de soustraction DCB qui l'effectue en une seule opération ! Naturellement, comme dans l'exemple précédent, le programme commencera par les instructions SED (qui met en mode décimal, à moins que cela n'ait été fait auparavant) et SEC qui met la retenue à 1. Ainsi, le programme pour soustraire le nombre DCB « 25 » du nombre DCB « 26 » est le suivant :

```
SED             METTRE EN MODE DECIMAL
SEC             METTRE LA RETENUE A1
LDA   #$26      CHARGER 26 EN DCB
SBC   #$25      MOINS 25
STA   ADR       RANGER LE RESULTAT
```

Addition DCB sur 16 bits

L'addition 16 bits s'effectue tout aussi simplement que dans le cas du binaire. Le programme d'une telle addition se trouve ci-dessous :

```
CLC
SED
LDA    ADR1
ADC    ADR2
STA    ADR3
LDA    ADR1-1
ADC    ADR2-1
STA    ADR3-1
```

Exercice 3.7 : *Comparer le programme ci-dessus avec celui de l'addition binaire sur 16 bits. Quelle est la différence ?*

Exercice 3.8 : *Ecrire un programme de soustraction DCB sur 16 bits (ne pas utiliser CLC et ADC !).*

Indicateurs DCB

En mode DCB, la présence d'un bit de retenue mis à 1 après une addition indique que le résultat est supérieur à 99. Ce n'est pas comme en complément à 2 puisque les chiffres DCB sont représentés en binaire sans signe. Inversement, l'absence de retenue lors d'une soustraction indique un emprunt.

Conseils pour l'addition et la soustraction

— Annulez toujours la retenue avant d'effectuer une addition.
— Mettez toujours le bit de retenue à 1 avant d'effectuer une soustraction.
— Spécifiez le mode voulu : binaire ou décimal.

Types d'instructions

Nous avons maintenant utilisé trois types d'instructions. Nous avons utilisé LDA et STA qui, respectivement, chargent l'accumulateur à partir d'une adresse-mémoire et rangent son contenu à l'adresse indiquée. Ces deux instructions sont des instructions de *transfert de données.*

Ensuite, nous avons utilisé des instructions *arithmétiques* telles que ADC et SBC. Elles effectuent respectivement une opération d'addition ou de soustraction. D'autres instructions de l'UAL vont être présentées dans ce chapitre.

Enfin, nous avons utilisé des instructions telles que CLC, SED et d'autres qui manipulent les bits indicateurs d'état (respectivement la

retenue et le mode décimal dans nos exemples). Ce sont les instructions de *manipulation des états* ou instructions de *contrôle.* Une description complète des instructions du 6502 sera présentée au Chapitre IV.

D'autres types d'instructions sont encore disponibles dans le microprocesseur, que nous n'avons pas encore utilisés. Ce sont en particulier les instructions de « saut » et de « branchement » qui modifient l'ordre d'exécution du programme. Ce nouveau type d'instructions va être introduit dans notre prochain exemple.

Multiplication

Examinons maintenant un problème arithmétique plus complexe : la multiplication de deux nombres binaires. Pour introduire l'algorithme de la multiplication binaire, commençons par examiner une multiplication décimale habituelle : nous allons multiplier 12 par 23.

```
    12  (multiplicande)
×   23  (multiplicateur)
    36  (Produit partiel)
+ 24
= 276  Résultat final.
```

La multiplication s'effectue en multipliant le chiffre le plus à droite du multiplicateur par le multiplicande soit « 3 » × « 12 ». Le produit partiel est « 36 ». Ensuite, on multiplie le chiffre suivant du multiplicateur, c'est-à-dire « 2 » par « 12 ». « 24 » est alors ajouté au produit partiel.

Mais il y a une opération de plus : 24 est *décalé* d'un cran vers la *gauche.* On pourrait dire de façon équivalente que le produit partiel (36) a été *décalé d'un cran vers la droite* avant l'addition.

Les deux nombres, correctement décalés, sont alors ajoutés et la somme est 276. C'est simple. Regardons maintenant la multiplication binaire. Elle est effectuée exactement de la même manière.

Examinons le sur un exemple. Multiplions 5 par 3 :

```
(5)      101    (MPD)
(3)   ×  011    (MPR)
        -----
         101    (PP)
        101
       000
        -----
(15)    01111   (RES)
```

Pour effectuer la multiplication binaire, nous opérons exactement comme ci-dessus en décimal. La représentation formelle de cet algo-

rithme apparaît sur la Figure 3-8. C'est l'ordinogramme de l'algorithme, notre premier ordinogramme.

Examinons le de plus près.

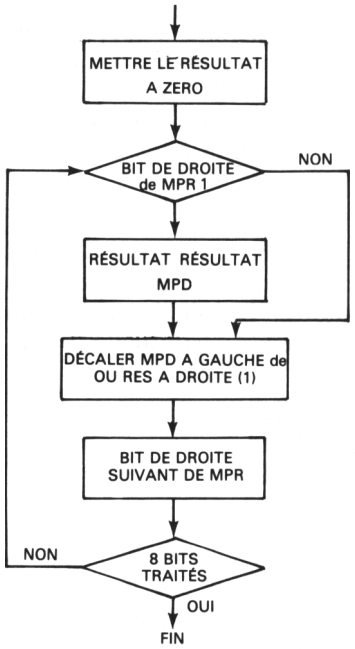


Fig. 3-8 : Algorithme de base de la multiplication : ordinogramme

Cet ordinogramme est la représentation symbolique de l'algorithme que nous venons de présenter. Chaque rectangle représente un ordre à exécuter. Il sera traduit en une ou plusieurs instructions du programme. Chaque bloc en forme de losange représente un test. Il correspondra à un branchement dans le programme. Si le test est positif, on se branchera à un endroit déterminé. Si le test est négatif, on se branchera à un autre endroit. La notion de branchement sera expliquée plus tard, dans le programme proprement dit. Le lecteur doit maintenant examiner l'ordinogramme et s'assurer qu'il représente bien l'algorithme exact tel qu'il a été présenté. Noter qu'il y a une flèche venant du dernier losange en bas de l'ordinogramme, qui remonte au premier losange en haut. C'est parce que la même portion de l'ordinogramme doit être effectuée huit fois, une fois pour chaque bit du multiplicateur. Une telle situation, où l'exécution recommence au même point s'appelle, pour des raisons évidentes, *une boucle de programme.*

Exercice 3.9 : *Multipliez « 4 » par « 7 » en binaire en utilisant l'ordinogramme et vérifiez que vous obtenez « 28 ». Sinon, réessayez. Ce n'est que si vous obtenez le résultat correct que vous êtes prêt à traduire cet ordinogramme en programme.*

Traduisons maintenant cet ordinogramme en programme, pour le 6502. Le programme complet apparaît sur la Figure 3-9. Nous allons maintenant l'étudier en détail. Comme vous vous en souvenez depuis le Chapitre 1, la programmation consiste ici à traduire l'ordinogramme de la Figure 3-8 pour obtenir le programme de la Figure 3-9. Chaque bloc de l'ordinogramme sera traduit en une instruction ou plusieurs. On supposera que MPR et MPD ont déjà une valeur.

```
        LDA #0          ANNULER L'ACCUMULATEUR
        STA TEMP        ANNULER CETTE ADRESSE
        STA RESAD       ANNULER
        STA RESAD +1    ANNULER
        LDX #8          X SERT DE COMPTEUR
MULT    LSR MPRAD       DECALER MPR A DROITE
        BCC NOADD       TESTER LE BIT DE RETENUE
        LDA RESAD       CHARGER A AVEC LA PARTIE
                        BASSE DE RES
        CLC             PREPARER L'ADDITION
        ADC MPDAD       AJOUTER MPD A RES
        STA RESAD       RANGER LE RESULTAT
        LDA RESAD +1    AJOUTER LE RESTE DE MPD
                        DECALE
        ADC TEMP
        STA RESAD +1
NOADD   ASL MPDAD       DECALER MPD A GAUCHE
        ROL TEMP        SAUVER LE BIT VENANT
                        DE MPD
        DEX             DECREMENTER LE COMPTEUR
        BNE MULT        RECOMMENCER
                        SI COMPTEUR = 0
```

Fig. 3-9 : Multiplication 8 par 8

Le premier bloc de l'ordinogramme est un bloc d'*initialisation*. Il est nécessaire de mettre à « 0 » un certain nombre de registres ou d'emplacements-mémoire car le programme s'en sert. Les registres qui sont utilisés par le programme de multiplication apparaissent sur la Figure 3-10. Sur la gauche de l'illustration apparaissent les régions concernées du microprocesseur 6502. Sur la droite de l'illustration, apparaît la portion de la mémoire utilisée. Nous suppose-

rons ici que les adresses-mémoire croissent du haut vers le bas de la figure. On pourrait naturellement utiliser la convention inverse. Tout à fait à gauche, le registre X (un des deux registres index du 6502) servira de *compteur*. Comme nous effectuons une multiplication 8 bits, nous aurons à tester les 8 bits du multiplicateur.

Malheureusement le 6502 ne possède pas d'instruction qui permette de tester ces bits à la suite. Les seuls bits qu'on puisse tester commodément sont les indicateurs dans le registre d'état. Il résulte de cette limitation, commune à la plupart des microprocesseurs, que pour tester successivement tous les bits du multiplicateur, il sera nécessaire de transférer la valeur du multiplicateur dans l'accumulateur. Ensuite, le contenu de l'accumulateur sera décalé à droite. Les instructions de décalage déplacent chaque bit du registre d'une position vers la droite ou vers la gauche. Le bit qui sort du registre tombe dans le bit de retenue « C » du registre d'état. L'effet des opérations de décalage est illustré sur la Figure 3-11. Il existe diverses variations possibles selon le bit qui est introduit dans le registre, et ces différences seront discutées au Chapitre IV (Jeu d'instructions du 6502).

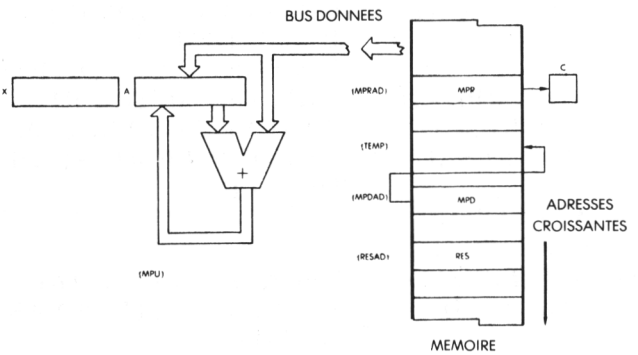


Fig. 3-10 : Multiplication : les registres

Revenons au test successif de chacun des 8 bits du multiplicateur. Comme on peut facilement tester le bit de retenue, le multiplicateur sera 8 fois décalé d'un cran. A chaque fois son bit le plus à droite tombera dans le bit de retenue où il sera testé.

Le problème suivant à résoudre est que le produit partiel à accumuler au cours des additions successives va nécessiter 16 bits. La multiplication de deux nombres de 8 bits peut donner un résultat de 16 bits. Ceci est dû au fait que $2^8 \times 2^8 = 2^{16}$. Nous devons réserver 16 bits pour ce résultat.

Malheureusement le 6502 a très peu de registres internes, de sorte que ce produit partiel ne peut-être stocké dans le 6502 lui-même. En fait, à cause du nombre limité de registres, nous ne pouvons même pas stocker le multiplicateur, le multiplicande ou le produit partiel à l'intérieur du 6502. Ils seront tous stockés en mémoire. Il en résultera une exécution plus lente que s'il avait été possible de tous les stocker dans des registres internes. C'est une limitation inhérente au 6502. La zone-mémoire utilisée pour la multiplication apparaît sur la droite de la Figure 3-10. En haut, apparaît le mot (octet) alloué au multiplicateur. Nous supposerons, par exemple, qu'il contient « 3 » en binaire. L'adresse de cette case-mémoire est MPRAD. En dessous, nous trouvons une « zone temporaire » dont l'adresse est TEMP. Le rôle de cet emplacement sera expliqué ci-dessous. Nous décalerons le multiplicande à gauche, dans cette case, avant de l'ajouter au produit partiel. Le multiplicande occupe la case suivante et on suppose qu'il contient la valeur « 5 » en binaire. Son adresse est MPDAD.

Enfin au bas de la mémoire nous trouvons les deux mots alloués au produit partiel ou au résultat. Leur adresse est RESAD.

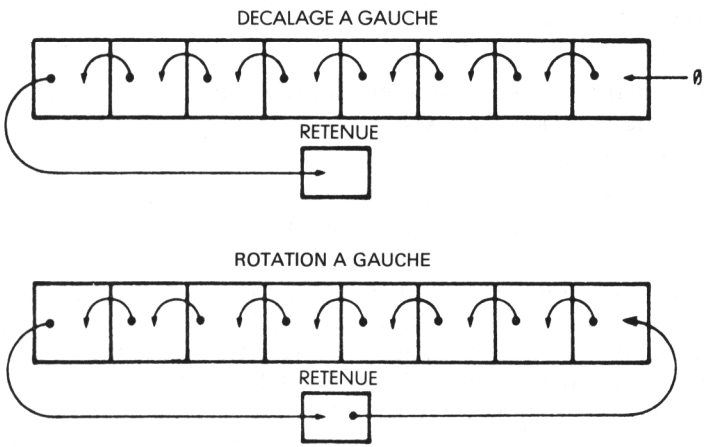


Fig. 3-11 : Décalage et rotation

Ces cases-mémoire formeront nos « registres de travail » et le mot « registre » peut être employé de façon interchangeable avec « case-mémoire » dans ce contexte.

La flèche qui apparaît en haut à droite de l'illustration et qui va de MPR au bit C est une manière symbolique de montrer comment le

multiplicateur sera décalé dans le bit de retenue. Bien entendu, ce bit de retenue est physiquement situé dans le 6502 et non dans la mémoire.

Revenons au programme de la Figure 3-9. Les cinq premières instructions sont des initialisations. Les quatre premières mettent à zéro les registres TEMP, RESAD, et RESAD + 1.

Vérifions le : LDA #0.

Cette instruction charge l'accumulateur avec la valeur littérale « 0 ». Il en résulte que l'accumulateur contiendra « 00000000 ».

Le contenu de l'accumulateur va maintenant servir à mettre à 0 les trois « registres » en mémoire. Il faut se rappeler que lire une valeur dans un registre ne le vide pas. Il est possible de lire un registre autant de fois que nécessaire. Cela ne change pas son contenu.

Continuons : STA TEMP

Cette instruction range le contenu de l'accumulateur dans la case-mémoire TEMP. Voir la Figure 3-10 pour suivre le cheminement des données dans le système. L'accumulateur contient « 00000000 ». Le résultat de cette instruction va être de mettre à zéro le contenu de l'adresse-mémoire TEMP. Rappelez-vous que le contenu de l'accumulateur reste 0 après une lecture. Il n'est pas changé. Nous allons l'utiliser à nouveau.

STA RESAD

Cette instruction opère comme la précédente et elle met à zéro le contenu de l'adresse RESAD. Faisons le encore une fois :

STA RESAD + 1

Enfin on met à zéro la case RESAD + 1 qui avait été réservée pour contenir la partie haute du résultat. (La partie haute est formée des bits 8 à 15, la partie basse des bits 0 à 7).

Enfin, pour pouvoir arrêter de décaler les bits du multiplicateur au bon moment, il est nécessaire de compter le nombre de décalages effectués. Il faut huit décalages. Le registre X est utilisé comme compteur et initialisé à la valeur « 8 ». Chaque fois qu'un décalage est effectué, le contenu de ce compteur est décrémenté de 1. Quand la valeur du compteur atteint « 0 », la multiplication est terminée. Initialisons le registre à « 8 » :

LDX #8

Cette instruction charge la constante « 8 » dans le registre X. D'après l'ordinogramme de la Figure 3-8, il faut tester le bit le moins significatif du multiplicateur. Il a été indiqué ci-dessus que ce test ne peut être fait en une seule instruction. Il faut utiliser deux instructions. On va d'abord décaler le multiplicateur vers la droite, pour tester le bit qui sera sorti lors du décalage. C'est le bit de retenue.

Effectuons ces opérations :

LSR MPRAD

Cette instruction est un *décalage logique à droite* du contenu de l'adresse mémoire MPRAD.

Exercice 3.10 : *Supposant que le multiplicateur est « 3 » dans notre exemple quel est le bit qui sort à droite de la case-mémoire MPRAD ? (En d'autres termes, quelle sera la valeur du bit de retenue après ce décalage ?).*

L'instruction suivante teste la valeur du bit de retenue :

BCC NOADD

Cette instruction veut dire « Branchement si retenue à zéro » à l'adresse NOADD.

C'est la première fois que nous rencontrons une instruction de branchement. Tous les programmes que nous avons considérés jusqu'ici étaient strictement séquentiels. Chaque instruction était exécutée après la précédente. Pour pouvoir utiliser des tests logiques comme celui du bit de retenue, il faut être capable d'aller, après le test, exécuter des instructions n'importe où dans le programme. C'est précisément le rôle joué par l'instruction de branchement. Elle teste la valeur du bit de retenue, alors le programme va se brancher à l'adresse NOADD. Cela signifie que la prochaine instruction exécutée après le BCC sera l'instruction à l'adresse NOADD si le test est positif.

Sinon, si le test est négatif, il n'y aura pas de branchement et c'est l'instruction qui suit BCC NOADD qui sera normalement exécutée.

Il faut encore donner une explication au sujet de « NOADD » : il s'agit d'une *étiquette symbolique* ou *label.* Elle représente sous forme symbolique une adresse physique effective dans la mémoire. Pour faciliter l'écriture du programme, l'assembleur permet d'utiliser des adresses symboliques au lieu d'adresses réelles. Lors de l'assemblage, l'assembleur substituera l'adresse physique proprement dite à la place du symbole « NOADD ». Cela améliore considérablement la lisibilité du programme et permet en outre au programmeur d'insérer des instructions supplémentaires entre le point de branchement et NOADD sans avoir à tout réécrire. Ces avantages seront étudiés plus en détail dans le Chapitre 10 sur l'assembleur.

Si le test est négatif, c'est l'instruction qui suit dans le programme qui est exécutée.

Nous allons maintenant étudier les deux cas :

Premier cas : la retenue était à 1

Si la retenue est 1, le test spécifié dans BCC est négatif et on exécute l'instruction qui suit BCC.

Deuxième cas : la retenue était à 0

Le test est positif et la prochaine instruction exécutée est celle qui se trouve à l'adresse NOADD.

D'après la Figure 3-8, l'ordinogramme spécifie que si le bit de retenue était à 1, il faut ajouter le multiplicande au produit partiel (ici les registres RES). De plus il faut effectuer un décalage. Le produit partiel doit être décalé d'un cran vers la droite, ou bien il faut décaler le multiplicande vers la gauche. Nous adoptons ici la convention habituelle utilisée lorsqu'on effectue une multiplication à la main, et nous décalerons le multiplicande d'un cran à gauche.

Le multiplicande est contenu dans les registres TEMP et MPDAD. (Pour simplifier nous donnons aux cases-mémoire le nom usuel de « registres »). Les 16 bits du produit partiel sont contenus aux adresses-mémoire RESAD et RESAD + 1. Pour illustrer ceci, supposons que le multiplicande était « 5 ». Les différents registres sont montrés sur la Figure 3-10.

Nous avons simplement à ajouter deux nombres de 16 bits. C'est un problème que nous avons appris à résoudre (si vous avez des doutes, reportez-vous à la section sur l'addition 16 bits, plus haut). Nous allons d'abord ajouter les octets d'ordre le plus bas puis les plus significatifs. Faisons le :

LDA RESAD

On charge l'accumulateur avec la partie basse de RES.

CLC

Avant toute addition, le 6502 exige que le bit de retenue soit annulé. Il est important de le faire ici puisque nous savons que le bit de retenue était 1. Il faut l'annuler.

ADC MPDAD

On ajoute le multiplicande à l'accumulateur qui contient maintenant RES bas.

STA RESAD

Le résultat de l'addition est rangé dans la case-mémoire convenable (RES bas). La première partie de l'addition est maintenant faite. Lorsque, plus tard, vous vérifierez ce programme à la main, n'oubliez pas que l'addition positionne le bit de retenue. La retenue sera mise à 0 ou à 1 selon les résultats produits. La retenue qui peut avoir été générée sera automatiquement reportée dans la partie haute du résultat final. Terminons l'addition :

LDA RESAD + 1

ADC TEMP

STA RESAD + 1

Ces trois instructions complètent notre addition 16 bits. Nous avons maintenant ajouté le multiplicande à RES. Il nous reste à le décaler d'un cran à gauche en vue de la prochaine addition.

Nous pourrions aussi avoir envisagé de décaler le multiplicande *avant* l'addition, sauf la première fois. C'est l'une des nombreuses options qui sont toujours offertes au programmeur.

Décalons le multiplicande à gauche :

```
NOADD ASL MPDAD
```

Cette instruction est un décalage arithmétique gauche. Elle décale d'un cran vers la gauche le contenu de la case-mémoire MPDAD qui se trouve contenir la partie basse du multiplicande. Cela ne suffit pas. Nous ne pouvons pas nous permettre de perdre le bit qui sort sur la gauche du multiplicande. Ce bit tombe dans l'indicateur de retenue. Il ne doit pas y être gardé en permanence car il risque d'être détruit par n'importe quelle opération arithmétique. Il faut le sauver dans un registre « permanent ». Il faudrait le rentrer dans la case mémoire TEMP. C'est précisément accompli par l'instruction suivante :

```
ROL TEMP
```

Elle spécifie « rotation à gauche » du contenu de TEMP.

Il convient d'observer ici un point intéressant. Nous venons d'utiliser deux sortes d'instructions de décalage différentes pour décaler un registre d'un cran vers la gauche. La première est ASL, la seconde ROL.

Quelle différence y a t-il ?

L'instruction ASL *décale* le contenu du registre. L'instruction ROL est une instruction de *rotation.* Elle décale le contenu du registre d'un cran vers la gauche, et le bit qui sort à gauche va dans la retenue comme d'habitude. La différence est que le *contenu antérieur du bit de retenue est entré dans la position la plus à droite.* En mathématique cela s'appelle une permutation circulaire (sur 9 bits). C'est exactement ce que nous voulons. Le ROL a pour résultat que le bit qui était sorti à gauche de MPD, et qui était préservé dans le bit de retenue C, va atterrir dans la position la plus à droite du registre TEMP.

Cela marche.

Nous en avons maintenant terminé avec la partie arithmétique de ce programme. Il nous faut encore tester si nous avons effectué les opérations 8 fois, c'est-à-dire si nous avons fini. Comme d'habitude

dans la plupart des microprocesseurs, ce test nécessite deux instructions :

DEX

Cette instruction décrémente le contenu du registre X. S'il contenait 8, il contiendra 7 après exécution de l'instruction.

BNE MULT

C'est une autre instruction de test et branchement. Elle signifie « branchement à l'adresse MULT si le résultat n'est pas égal à zéro ». Tant que notre compteur décrémente pour donner un entier non nul, on branchera automatiquement à l'adresse MULT.

Il s'agit de la boucle de multiplication. En se reportant à l'ordinogramme de la multiplication, ce branchement correspond à la flèche qui sort du dernier bloc. Cette boucle sera exécutée 8 fois.

Exercice 3.11 : *Qu'arrive-t-il lorsque la décrémentation de X donne « 0 » ? Quelle est la prochaine instruction exécutée ?*

Dans la plupart des cas, le programme que nous venons de développer sera un *sous-programme* et la dernière instruction du sous-programme sera un RTS. Le mécanisme des sous-programmes sera expliqué plus loin dans ce chapitre.

TEST PERSONNEL IMPORTANT

Si vous souhaitez apprendre à programmer, il est extrêmement important que vous compreniez un programme typique, de ce genre, dans tous les détails. Nous avons introduit beaucoup d'instructions nouvelles. L'algorithme est raisonnablement simple mais le programme est bien plus long que les programmes que nous avions développés jusque-là. *Il vous est fortement recommandé de faire l'exercice suivant entièrement et correctement avant de continuer ce chapitre.* Si vous le faites correctement, vous aurez réellement compris par quels mécanismes les instructions manipulent le contenu de la mémoire et des registres du microprocesseur et comment est utilisé l'indicateur de retenue. Si vous ne le faites pas, il est probable que vous aurez des difficultés lorsque vous écrirez, vous-même, des programmes. L'apprentissage de la programmation nécessite que vous programmiez vous-même. S'il vous plaît, prenez une feuille de papier et faites l'exercice suivant :

Exercice 3.12 : *Chaque fois qu'un programme est écrit, on devrait le vérifier à la main pour s'assurer que ses résultats seront corrects. Nous allons précisément faire cela : le but de cet exercice est de remplir le tableau de la Figure 3-12.*

LABEL	INSTRUCTION	X	A	MPR	C	TEMP	MPD	(RESAD)L	(RESAD)H

Fig. 3-12 : Tableau à remplir pour l'exercice 3-12

Vous pouvez écrire dessus directement, ou bien le recopier. Le but est de déterminer le contenu de chaque registre et adresse-mémoire concernés après l'exécution de chaque instruction du programme, et ce, du début à la fin.

Vous trouverez sur une horizontale de la Figure 3-12 tous les registres utilisés par le programme : X, A, MPR, C (bit de retenue), TEMP, MPD, RESADL (résultat bas), RESADH. A gauche, vous devez mettre le label s'il y en a un, et l'instruction en cours d'exécution. A droite de l'instruction, vous devez écrire le contenu de tous les registres après exécution de l'instruction. Quand le contenu d'un registre est indéfini, mettez des tirets. Commençons à remplir ce tableau ensemble.

La première ligne apparaît ci-dessous :

ETIQUETTE	INSTRUCTION	X	A	MPR	C	TEMP	MPD	RESAD)L	(RESAD)H
	LDA #0	-----	00000000	00000011	--	-----	00000101	-----	-----

Fig. 3-13 : Première instruction pour la multiplication

La première instruction à exécuter est LDA #0.

Après son exécution, le contenu du registre X est inconnu. On utilise des tirets pour l'indiquer. L'accumulateur ne contient que des zéros. Nous supposons aussi que le multiplicateur et le multiplicande avaient été chargés avec une valeur par le programmeur avant l'exécution de ce programme. (Sinon, il aurait fallu des instructions supplémentaires pour établir le contenu de MPD et MPR). Nous trouvons dans MPR la représentation binaire de la valeur 3 et dans MPD celle de 5. La retenue est indéfinie. Le registre TEMP est indéfini et les deux registres utilisés pour RES sont indéfinis. Remplissons maintenant la ligne suivante : elle apparaît ci-dessous : la seule différence est que le contenu de TEMP a été mis à 0. L'instruction suivante mettra le contenu de RESAD à 0 et la suivante mettra à 0 celui de RESAD + 1.

ETIQUETTE	INSTRUCTION	X	A	MPR	C	TEMP	MPD	(RES)L	(RES)H
	LDA #0	-----	00000000	00000011	--	-----	00000101	-----	-----
	STA TEMP					00000000			

Fig. 3-14 : Les deux premières lignes de la multiplication

La cinquième instruction : LDX #8 met le contenu de X à « 8 ».

Examinons encore une instruction ensemble (voir Figure 3-15).

L'instruction LSR MPRAD décale le contenu de MPRAD d'un cran vers la droite. Vous pouvez voir qu'après le décalage, le contenu de MPR est « 00000001 ». Le 1 le plus à droite de MPR est tombé dans le bit de retenue. Le bit C est maintenant à 1. Les autres registres sont inchangés.

ETIQUETTE	INSTRUCTION	X	A	MPR	C	TEMP	MPD	(RESAD)L	(RESAD)H
	LDA #0	------	00000000	00000011	--	------	00000101	------	------
	STA TEMP					00000000			
000	STA RESAD							00000000	
	STA RESAD + 1								00000000
	LDX #8	00001000							
MULT	LSR MPRAD			00000001	1				
	BCC NOADD								
	LDA RESAD								
	CLC				0				
	ADC MPDAD		00000101						
101	STA RESAD							00000101	
	LDA RESAD + 1		00000000						
	ADC TEMP								
	STA RESAD + 1								
NOADD	ASL MPDAD						00001010		
	ROL TEMP								
	DEX	00000111							
	BNE MULT								
MULT	LSR MPRAD			00000000	1				
		(2nd ITERATION)							

Fig. 3-15 : Tableau de l'exercice 3.12 partiellement rempli

C'est maintenant votre tour. Remplissez le reste du tableau complètement. Ce n'est pas difficile mais il faut faire attention. Si vous avez des doutes sur l'effet de certaines instructions, vous pouvez vous reporter au Chapitre 4 où vous trouverez leur description, ou aux appendices où elles sont listées sous forme de tableaux.

Le résultat final de votre multiplication devrait être « 15 » en binaire, dans les registres RESAD haut et bas. RESAD haut doit être mis à « 00000000 ». RESAD bas doit être « 00001111 ». Si vous avez obtenu ce résultat, vous avez gagné. Sinon, réessayez. La source d'erreurs la plus fréquente est une mauvaise gestion du bit de retenue. Assurez-vous que le bit de retenue est bien correct chaque fois que vous effectuez une opération arithmétique. N'oubliez pas que l'UAL établit le bit de retenue après chaque addition.

Alternatives de programmation

Le programme que nous venons de développer n'est qu'une des nombreuses formes sous lesquelles il pourrait avoir été écrit. Tout programmeur peut trouver des façons de modifier, et parfois d'améliorer, un programme. Par exemple, nous avons décalé à gauche le multiplicande avant l'addition. Il aurait été mathématiquement équivalent de décaler le résultat à droite avant de l'ajouter au multiplicande. L'avantage est qu'on n'aurait pas eu besoin du registre

TEMP, ce qui aurait économisé une case-mémoire. Cette méthode serait préférée dans un microprocesseur muni d'assez de registres internes pour garder MPR, MPD et RESAD à l'intérieur du microprocesseur. Comme nous avons été obligés d'utiliser la mémoire pour cela, l'économie d'une case-mémoire n'a pas d'intérêt. La question est, par suite, de savoir si la seconde méthode peut entraîner un gain en vitesse de la multiplication. C'est un exercice intéressant :

Exercice 3-13 : *Ecrivez maintenant une multiplication 8 × 8 à l'aide du même algorithme mais en décalant le résultat d'un cran à droite au lieu de décaler le multiplicande d'un cran à gauche. Comparez-le au programme précédent et déterminez si cette nouvelle approche est plus rapide ou plus lente que la précédente.*

Un autre problème peut être soulevé : pour déterminer la vitesse du programme, nous pouvons souhaiter nous reporter aux tables de l'appendice qui listent le nombre de cycles demandés par chaque instruction. Mais le nombre de cycles demandés par certaines instructions dépend de l'endroit où elles sont situées. Le 6502 a un mode d'adressage particulier appelé *adressage direct* ou en *page 0*, dans lequel on réserve la première page (adresses-mémoire 0 à 255) pour les exécutions rapides. Cela sera expliqué au Chapitre 5 sur les techniques d'adressage. Disons brièvement que les programmes qui doivent s'exécuter rapidement ont leurs variables implantées en page 0 de sorte que les instructions ne nécessitent que deux octets pour adresser une case-mémoire (l'adressage de 256 cases ne demande qu'un octet) alors que des variables implantées ailleurs en mémoire ont besoin généralement d'instructions sur 3 octets. Reportons cette analyse au Chapitre V.

Programme de multiplication amélioré

Le programme que nous venons de développer est une traduction évidente de l'algorithme. Cependant, la programmation efficace exige qu'on porte une attention détaillée afin de réduire la longueur du programme et améliorer sa vitesse d'exécution. Nous allons maintenant présenter une réalisation améliorée du même algorithme.

Une des tâches qui consomment des instructions et du temps est le décalage du résultat et du multiplicateur. Une « astuce » habituellement utilisée dans l'algorithme de multiplication est fondée sur l'observation suivante : chaque fois que le multiplicateur est décalé d'un rang vers la droite, un bit devient libre sur la gauche. En même temps, on peut observer que le premier résultat (ou produit partiel) occupera au plus 9 bits. Après le décalage de multiplication suivant, la taille du produit partiel s'accroîtra encore d'un bit. En d'autres

termes, nous pouvons ne réserver au départ qu'un emplacement-mémoire pour le produit partiel, et utiliser les positions libérées au fur et à mesure que le multiplicateur est libéré.

Nous allons maintenant décaler le multiplicateur sur la droite. Il va libérer une position à sa gauche et nous entrerons le bit le plus à droite du produit partiel dans la position qui vient d'être libérée. Considérons maintenant l'utilisation optimale des registres. La Figure 3-16 montre les registres internes du 6502. Le mieux est d'utiliser X comme compteur. (Il peut être décrémenté en une seule instruction, DEX). Nous l'utiliserons comme compteur des bits décalés. L'accumulateur est (malheureusement) le seul registre interne qui puisse être décalé. Pour augmenter l'efficacité, il faudrait y mettre soit le multiplicateur soit le résultat.

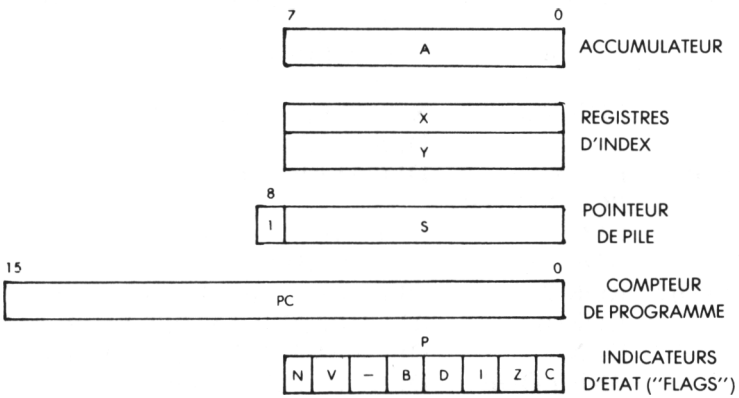


Fig. 3-16 : Registres du 6502

Lequel devons-nous mettre dans l'accumulateur ? Le résultat doit être ajouté au multiplicande à chaque fois qu'on sort un « 1 » du multiplicateur. Comme dans le 6502, on ne peut ajouter quelque chose qu'à l'accumulateur, c'est le résultat qui résidera dans l'accumulateur.

Les autres nombres devront résider en mémoire (cf. Figure 3-17) A et B vont contenir le résultat. A contiendra la partie haute du résultat tandis que B en contiendra la partie basse. A est l'accumulateur, B est un emplacement mémoire, préférablement en page 0. L'emplacement-mémoire C contient le multiplicateur. L'emplacement-mémoire D contient le multiplicande. Le programme apparaît ci-dessous.

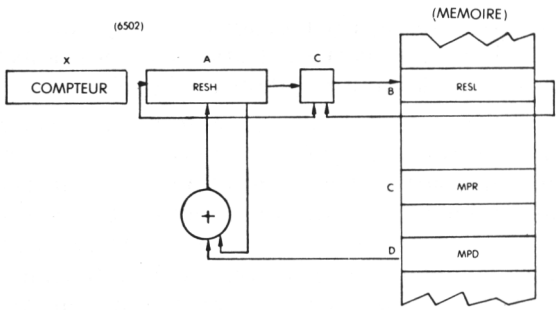


Fig. 3-17 : Allocation des registres dans la multiplication améliorée

```
MULT    LDA # 0       INITIALISATION DU RESULTAT
                      A 0 (HAUT)
        STA B         INITIALISATION DU RESULTAT
                      (BAS)
        LDX #8        X EST LE COMPTEUR
                      DES DECALAGES
LOOP    LSR C         DECALER MPR
        BCC NOADD
        CLC           RETENUE = 1 : L'ANNULER
        ADC D         A=A + MPD
NOADD   ROR A         DECALER LE RESULTAT
        ROR B         RECUPERER LE BIT EN B
        DEX           DECREMENTER LE COMPTEUR
        BNE LOOP      DERNIER DECALAGE ?
```

Fig. 3-18 : Multiplication améliorée

Examinons le programme. Comme A et B vont contenir le résultat, ils doivent être initialisés à 0. Faisons-le :

```
MULT LDA #0
     STA B
```

Nous utiliserons le registre X comme compteur de décalages et nous l'initialisons à la valeur 8.

```
LDX #8
```

Nous sommes maintenant prêts à entrer dans la boucle principale de multiplication comme précédemment. Nous allons d'abord déca-

ler le multiplicateur, puis tester le bit de retenue qui contient le bit le plus à droite tel qu'il est sorti du multiplicateur. Faisons-le :

```
LOOP LSR C
     BCC NOADD
```

Il y a deux possibilités : si la retenue était « 0 », nous nous brancherions à NOADD. Supposons qu'elle soit « 1 ». Nous continuons :

```
CLC
ADC D
```

Comme la retenue était à « 1 », elle doit être annulée et ensuite nous ajoutons le multiplicande à l'accumulateur. (L'accumulateur contient le résultat, « 0 » jusqu'ici).

Décalons alors le produit partiel :

```
NOADD ROR A
      ROR B
```

Le produit partiel en A est décalé à droite de une position. Le bit le plus à droite se retrouve dans la retenue. Le bit de retenue est récupéré et entré par rotation dans le registre (mémoire) B qui contient la partie basse du résultat.

Nous n'avons plus qu'à tester si nous avons fini :

```
DEX
BNE LOOP
```

Si nous examinons ce nouveau programme, nous voyons qu'il est écrit en moitié moins d'instructions que le précédent. De plus, il s'exécutera beaucoup plus vite. Ceci montre l'intérêt de choisir les registres appropriés pour contenir les informations.

Une écriture de programme directe fournira un programme qui marche correctement, mais pas un programme optimisé. Il est donc d'importance considérable d'utiliser les registres disponibles et les mémoires de la meilleure façon possible. Cet exemple illustre une approche rationnelle au choix des registres, conduisant à l'efficacité maximum.

Exercice 3.14 : *Calculer la vitesse d'une multiplication utilisant ce dernier programme. On supposera que le branchement a lieu dans cinquante pour cent des cas. Regarder le nombre de cycles demandés par chaque instruction dans la table de la fin du livre, et supposer une vitesse d'horloge où 1 cycle = 1 microseconde.*

Division binaire

L'algorithme de la division binaire est analogue à celui qu'on a utilisé pour la multiplication. Le diviseur est successivement soustrait des bits d'ordre élevé du dividende. Après chaque soustraction,

on utilise le résultat au lieu du dividende initial. Le quotient est augmenté de « 1 » en même temps. Eventuellement, le résultat de la soustraction peut être négatif. Cela s'appelle un dépassement. On doit alors rétablir le résultat partiel en lui rajoutant le diviseur. Naturellement, le quotient doit en même temps être redécrémenté de « 1 ». Quotient et dividende sont alors décalés d'un cran à gauche et on recommence l'algorithme.

La méthode qu'on vient de décrire s'appelle *méthode avec rétablissement.* Une variante de cette méthode qui donne une vitesse d'exécution améliorée s'appelle *méthode sans rétablissement.*

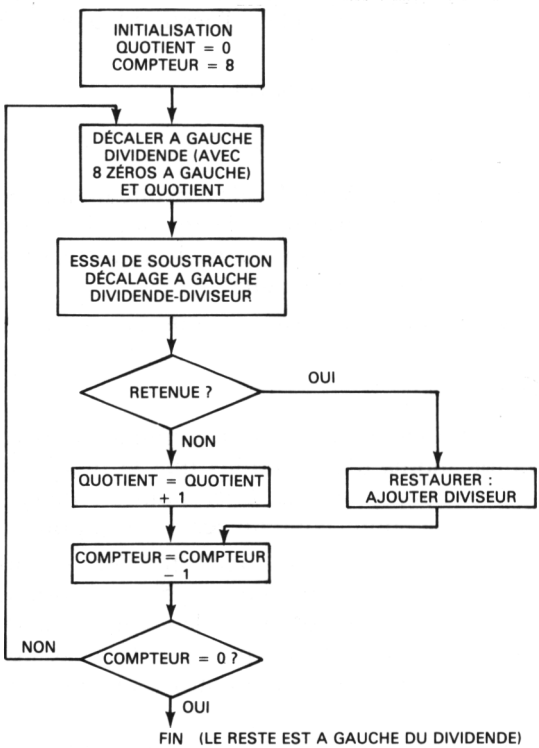


Fig. 3-19 : Ordinogramme de la division binaire 8 bits

La division 16 bits

Nous décrivons maintenant la division sans rétablissement pour un dividende 16 bits et un diviseur 8 bits. Le résultat a généralement 8 bits et le programme présenté ne fonctionne que si le résultat tient sur 8 bits. Les registres et l'implantation mémoire de ce programme sont montrés Figure 3-20. Le dividende est contenu dans l'accumu-

lateur (partie haute) et dans la case mémoire 0, désignée ici sous le nom B. Le résultat est en Q (adresse 1). Le diviseur est dans D (adresse 2). Le reste se retrouvera dans l'accumulateur A.

Le programme apparaît Figure 3-21 et l'ordinogramme correspondant est sur la Figure 3-22.

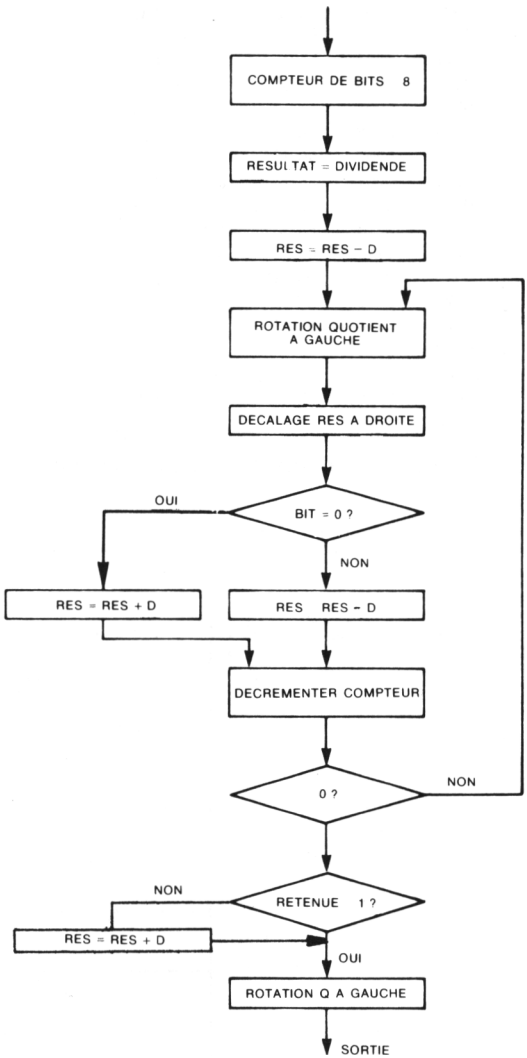


Fig. 3-20 : Ordinogramme de la division 16 par 8

```
No. LIGNE  ADR   CODE      LIGNE
0002       0000                *  = $0
0003       0000              B *  = * + 1
0004       0001              Q *  = * + 1
0005       0002              D *  = * + 1
0006       0003                *  = $200
0007       0200  A0 08     DIV   LDY #8
0008       0202  38              SEC
0009       0203  E5 02           SBC D
0010       0205  08        LOOP  PHP
0011       0206  26 01           ROL Q
0012       0208  06 00           ASL B
0013       020A  2A              ROL A
0014       020B  28              PLP
0015       020C  90 05           BCC ADD
0016       020E  E5 02           SBC D
0017       0210  4C 15 02        JMP NEXT
0018       0213  65 02     ADD   ADC D
0019       0215  88        NEXT  DEY
0020       0216  D0 ED           BNE LOOP
0021       0218  B0 03           BCS LAST
0022       021A  65 02           ADC D
0023       021C  18              CLC
0024       021D  26 01     LAST  ROL Q
0025       021F  00              BRK
0026       0220                  END
```

Fig. 3-21 : Programme de division 16 par 8

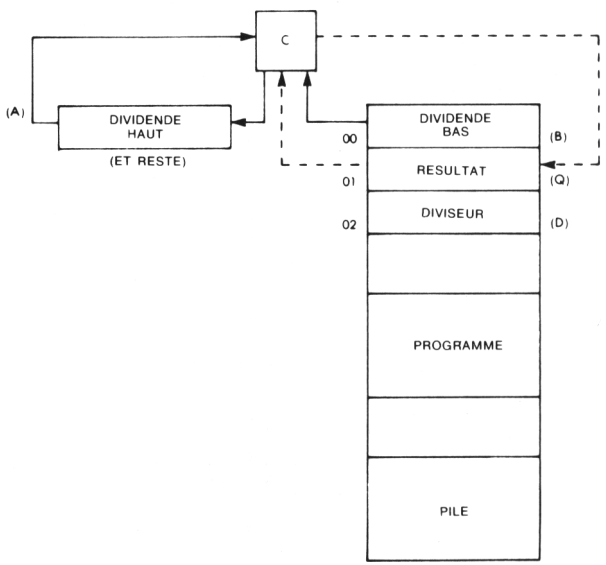


Fig. 3-22 : Implantation mémoire pour la division

Exercice 3-15 : *Vérifiez le fonctionnement de ce programme comme pour l'exercice 3-12. Par exemple 33 divisé par 3 doit donner 11 et un reste de 0.*

OPERATIONS LOGIQUES

L'autre catégorie d'instructions, qui, à l'intérieur du microprocesseur, peuvent être exécutées par l'UAL, est l'ensemble des instructions logiques. Elles comprennent AND (ET), ORA (OU) et le OU exclusif : EOR.

En outre on peut y inclure les opérations de décalage qui ont déjà été utilisées et l'instruction de comparaison qui s'appelle CMP sur le 6502. L'utilisation de AND, ORA, et EOR sera décrite dans le Chapitre IV qui présente l'ensemble des instructions.

Développons maintenant un court programme, qui teste si un emplacement mémoire, appelé LOC, contient la valeur « 0 », la valeur « 1 » ou une autre. Le programme apparaît ci-dessous.

```
        LDA   LOC     LIRE LE CARACTERE EN LOC
        CMP   #$00    COMPARER A ZERO
        BEQ   ZERO    EST-CE 0 ?
        CMP   #$01    1
        BEQ   UN
AUTRE   ...
ZERO    ...
UN      ...
```

Test de 0 ou 1

La première instruction : LDA LOC lit le contenu de la mémoire LOC.

C'est le caractère que nous voulons tester.

CMP #$00

Cette instruction compare le contenu de l'accumulateur avec la constante hexadécimale « 00 », c'est-à-dire le motif binaire « 00000000 ». Cette instruction de comparaison va positionner le bit Z (zéro) du registre d'état, qui va alors être testé par la prochaine instruction.

BEQ ZERO

L'instruction BEQ ZERO veut dire « branchement si égal ». L'instruction détermine si le test est positif en examinant le bit Z. Si le bit

est positionné, le programme se branche en ZERO. Si le test est négatif on exécute l'instruction en séquence :

CMP #$01

Le processus se répète en comparant par rapport au nouveau motif. Si le test est positif, l'instruction suivante amènera un branchement à l'adresse UN. Sinon, on exécutera l'instruction qui suit.

Exercice 3.16: *Ecrire un programme qui lise le contenu de l'adresse mémoire « 24 » et se branche à l'adresse appelée « STAR » s'il y avait une « * » à l'adresse 24. Le motif binaire correspondant à « * » est « 00101010 ».*

RECAPITULATION

Nous avons maintenant étudié et utilisé la plupart des instructions importantes du 6502. Nous avons transféré des valeurs entre la mémoire et les registres. Nous avons effectué des opérations arithmétiques et logiques sur ces données. Nous les avons testées et, selon les résultats de ces tests, nous avons exécuté différentes parties du programme. Nous avons aussi, dans le programme de multiplication, introduit une structure appelée boucle. Nous allons maintenant introduire une autre structure importante, le sous-programme.

SOUS-PROGRAMMES

Du point de vue conceptuel, un sous-programme est simplement un bloc d'instructions auquel le programmeur a donné un nom. Du point de vue pratique, un sous-programme doit commencer par une instruction spéciale dite *déclaration de sous-programme,* qui l'identifie comme sous-programme pour l'assembleur. Il se termine par une autre instruction spéciale appelée *retour.* Illustrons d'abord l'utilisation des sous-programmes dans un programme afin de montrer leur intérêt. Ensuite nous examinerons comment ceci est effectivement réalisé.

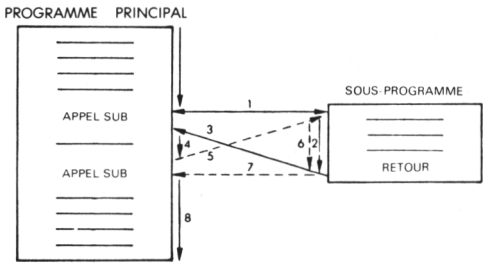


Fig. 3-23 : Appels de sous-programmes

L'utilisation d'un sous-programme est illustrée par la Figure 3-21. Le programme principal apparaît à gauche de l'illustration. Le sous-programme est représenté symboliquement sur la droite. Examinons le mécanisme des sous-programmes. Les lignes du programme principal sont exécutées en séquence jusqu'à ce qu'on rencontre une instruction intitulée APPEL SUB (CALL) en programmation. Cette instruction particulière est l'*appel* du sous-programme, et elle a pour effet un transfert au sous-programme. Cela signifie que la prochaine instruction exécutée après l'appel sera la première instruction située dans le sous-programme. Ceci est illustré par la flèche 1 sur la figure.

Ensuite le sous-programme s'exécute exactement comme tout autre programme. Nous supposerons que le sous-programme ne contient pas d'autre appel. La dernière instruction du sous-programme est un RETOUR (RETURN). C'est une instruction spéciale qui cause un retour au programme principal. La prochaine instruction qui sera exécutée après le RETOUR est celle qui suit l'APPEL SUB. C'est illustré par la flèche 3 sur la figure. L'exécution du programme principal continue alors, comme indiqué par la flèche 4. Dans le cours du programme principal apparaît un second APPEL SUB. Un nouveau transfert a lieu, représenté par la flèche 5. Cela signifie que le corps du sous-programme est exécuté une seconde fois à la suite de l'exécution d'APPEL SUB. Quand le retour du sous-programme est rencontré on retourne à l'instruction qui suit l'APPEL SUB concerné. Ceci est illustré par la flèche 7. Après le retour au programme principal l'exécution se poursuit normalement, comme le montre la flèche 8.

Le rôle des instructions spéciales APPEL SUB et RETOUR devrait maintenant être clair. Quel est l'intérêt du sous-programme ?

L'intérêt essentiel du sous-programme est qu'il peut être appelé depuis un nombre quelconque d'endroits du programme principal et utilisé plusieurs fois sans avoir à le réécrire. Un premier avantage est que cette approche économise de la place mémoire et il n'y a pas besoin de réécrire le sous-programme à chaque fois. Un second avantage est que le programmeur peut écrire un sous-programme donné une seule fois puis l'utiliser de façon répétée. C'est une simplification considérable de la conception des programmes.

Exercice 3.17 : *Quel est le principal inconvénient des sous-programmes ?*

L'inconvénient du sous-programme devrait apparaître clairement rien qu'en examinant le cours de l'exécution entre le programme principal et le sous-programme. Un sous-programme entraîne un ralen-

tissement de l'exécution puisqu'il y a des instructions supplémentaires à exécuter : L'APPEL SUB et le RETOUR.

Réalisation du mécanisme de sous-programme

Nous allons examiner ici comment les deux instructions spéciales APPEL SUB et RETOUR sont réalisées à l'intérieur du processeur. L'effet de l'instruction APPEL SUB est que la prochaine instruction soit recherchée à une nouvelle adresse. Vous vous rappelez (sinon, relisez le Chapitre 1) que l'adresse de la prochaine instruction à exécuter dans un ordinateur est contenue dans le compteur ordinal (PC). Cela signifie que l'effet de l'APPEL SUB est d'imposer un nouveau contenu au registre PC. Son effet est de charger dans le compteur ordinal, l'adresse de début du sous-programme. *Est-ce que cela suffit ?*. Pour répondre à cette question, considérons l'autre instruction qui doit être réalisée : le RETOUR. Le RETOUR doit entraîner, comme son nom l'indique, un retour à l'instruction qui suit l'APPEL SUB. Ce n'est possible que si l'adresse de cette instruction a été préservée quelque part. Cette adresse se trouve être la valeur du compteur ordinal au moment où l'on rencontre l'APPEL SUB. Ceci est dû au fait que le compteur ordinal est automatiquement incrémenté à chaque fois qu'on l'utilise (relisez le Chapitre 1 s'il le faut). C'est précisément l'adresse que nous voulons préserver pour pouvoir ultérieurement effectuer le RETOUR.

La question suivante est : où pouvons-nous préserver cette adresse de retour ? Cette adresse doit être préservée dans un emplacement raisonnable où il est garanti qu'elle ne sera pas effacée. Cependant considérons maintenant la situation illustrée par la Figure 3-24 : dans cet exemple le sous-programme 1 contient un appel à SUB2.

Il faut que notre mécanisme marche aussi bien dans ce cas. Naturellement il pourrait même y avoir plus de deux sous-programmes, disons N appels « imbriqués ». A chaque fois qu'un nouvel appel est rencontré, le mécanisme doit donc ranger à nouveau le contenu du compteur ordinal. Ceci implique le besoin d'au moins 2N emplacements-mémoire pour ce mécanisme. En outre il faut revenir d'abord de SUB2 puis de SUB1. En d'autres termes, nous avons besoin d'une structure qui conserve l'ordre chronologique dans lequel les données ont été sauvegardées. Cette structure a un nom. Nous l'avons déjà introduite. C'est la *pile*. La Figure 3-26 montre le contenu effectif de la pile au cours des appels de sous-programme successifs. Regardons d'abord le programme principal A l'adresse 100, on rencontre le premier appel : APPEL SUB 1.

Nous supposerons ici que l'appel de sous-programme occupe 3 octets (il pourrait en occuper deux). L'adresse qui suit en séquence

est, par suite, non pas « 101 », mais « 103 ». L'instruction d'appel occupe les adresses « 100 », « 101 » et « 102 ».

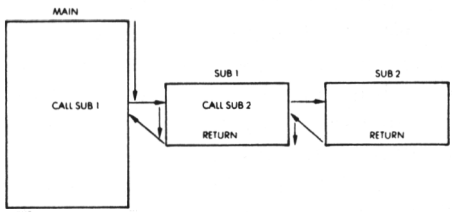


Fig. 3-24 : Appels imbriqués

Comme l'unité de commande du 6502 « sait » qu'il s'agit d'une instruction de 3 octets, la valeur du compteur ordinal quand l'appel aura été complètement décodé sera « 103 ». L'effet de l'appel sera de charger la valeur « 280 » dans le compteur ordinal. « 280 » est l'adresse de départ de SUB1.

Le second effet de l'APPEL sera d'empiler (préserver dans la pile) la valeur « 103 » du compteur ordinal. Ceci est illustré au bas à gauche de la Figure 3-26 qui montre qu'à l'instant 1, la valeur « 103 » est mise dans la pile. Allons plus à droite dans la Figure 3-25. A l'adresse 300 nous rencontrons un nouvel appel. Exactement comme dans le cas précédent, la valeur « 900 » sera chargée dans le compteur ordinal. C'est l'adresse de départ de SUB2. En même temps la valeur « 303 » sera introduite dans la pile. L'exécution va alors continuer sur la droite de la figure, dans SUB2. Nous sommes prêts maintenant à démontrer l'effet de l'instruction RETOUR et le fonctionnement correct de notre mécanisme de pile. L'exécution se poursuit dans SUB2 jusqu'à ce qu'on tombe sur l'instruction RETOUR à l'instant 3. L'effet de l'instruction RETOUR est simplement de desempiler le sommet de la pile et de l'envoyer dans le compteur ordinal. En d'autres termes, le compteur ordinal est restauré à la valeur qu'il avait avant l'entrée dans le sous-programme. Le sommet de la pile, dans notre exemple, vaut « 303 ». La Figure 3-26 montre que, à l'instant 3, la valeur « 303 » a été enlevée de la pile et remise dans le compteur ordinal. Il en résulte que l'exécution continue à partir de l'adresse « 303 ». A l'instant 4, on rencontre le RETOUR de SUB 1. La valeur du sommet de pile est « 103 ». Elle est extraite et placée dans le compteur ordinal. Il en résulte que l'exécution va se poursuivre à partir de l'adresse « 103 » dans le programme principal. C'est de fait, exactement l'effet que nous voulions. La Figure 3-26 montre qu'à l'instant 4, la pile est de nouveau vide. Le mécanisme fonctionne.

Le mécanisme d'appel des sous-programmes fonctionne jusqu'à

ce qu'on atteigne la dimension maximum de la pile. C'est pourquoi les premiers microprocesseurs, qui avaient une pile de 4 ou 8 registres, étaient limités à 4 ou 8 niveaux d'appels de sous-programmes.

En théorie, le 6502 qui est limité à 256 octets pour la pile (page 1) peut, par suite, accepter jusqu'à 128 appels de sous-programmes imbriqués. Ce n'est vrai que s'il n'y a pas d'interruptions, si la pile n'est pas utilisée pour autre chose, et s'il n'y a pas de registres autres que PC à préserver dans la pile. En pratique on utilise donc moins de niveaux de sous-programmes imbriqués.

Noter, sur les Figures 3-24 et 3-25, que les sous-programmes ont été représentés à droite du programme principal. C'est uniquement pour la clarté du dessin. En réalité, les instructions des sous-programmes sont écrites par l'utilisateur comme des instructions normales du programme. Sur une feuille de papier contenant le listing du programme complet, les sous-programmes peuvent être au début du texte, au milieu ou à la fin. C'est pourquoi ils sont précédés d'une déclaration de sous-programme : il est nécessaire de les identifier. Ces instructions spéciales disent à l'assembleur qu'il doit traiter ce qui suit comme un sous-programme. De telles *directives* d'assemblage seront présentées au Chapitre X.

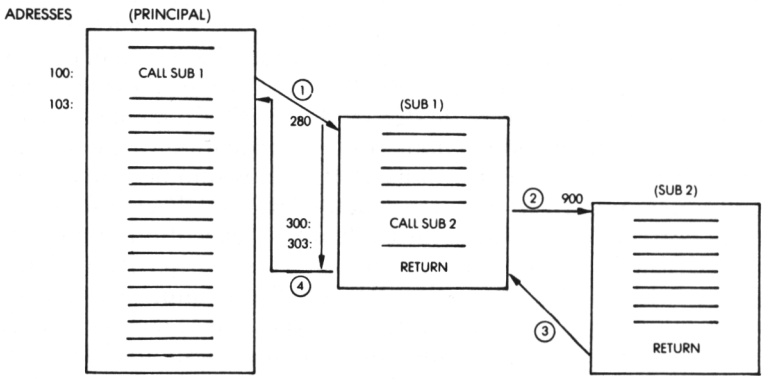


Fig. 3-25 : Les appels de sous-programmes

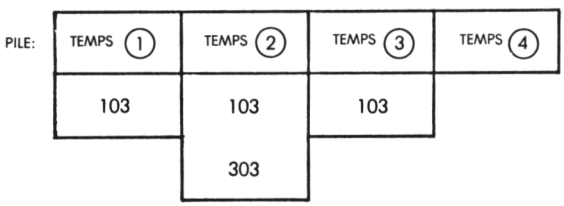


Fig. 3-26 : La pile en fonction du temps

Les sous-programmes sur le 6502

Nous avons maintenant décrit le mécanisme des sous-programmes et comment on fait appel à la pile pour le mettre en œuvre. L'instruction d'appel de sous-programme pour le 6502 s'appelle JSR (Jump to Sub Routine = saut à un sous-programme). C'est malheureusement un saut inconditionnel : il ne teste pas de bit d'état. Il faut insérer explicitement des branchements avant le JSR si l'on doit faire des tests. Le retour de sous-programme est l'instruction RTS (Retour de sous-programme). C'est une instruction sur 1 octet.

Exercice 3.18 : *Pourquoi le retour d'un sous-programme est-il tellement plus court que l'appel (Conseil : si la réponse n'est pas évidente, revoyez la mise en œuvre du mécanisme de sous-programme à l'aide de la pile et analysez les opérations internes à effectuer) !*

Exemples de sous-programmes

La plupart des programmes que nous avons développés ou allons développer seraient normalement écrits sous forme de sous-programmes. Par exemple, il est probable que le programme de multiplication soit utilisé en beaucoup d'endroits du programme. Pour faciliter le développement du programme, et le clarifier, il est par suite, commode de le définir comme sous-programme dont le nom serait, par exemple MULT. A la fin de ce sous-programme, nous aurions simplement à ajouter l'instruction RTS.

Exercice 3.19 : *Si on utilise MULT comme sous-programme, est-ce qu'il « détruit » des indicateurs d'état ou des registres internes ?*

Récursivité

La récursivité désigne le fait qu'un sous-programme s'appelle lui-même. Si vous avez compris la mise en œuvre du mécanisme, vous devriez maintenant être capable de répondre à la question suivante :

Exercice 3.20 : *Est-il légal de laisser un sous-programme s'appeler lui-même ? (En d'autres termes, est-ce que tout marchera, même si un sous-programme s'appelle lui-même ?). Si vous n'êtes pas sûr, dessinez la pile et remplissez la avec les adresses successives. Vous vérifierez physiquement si cela marche ou non. Cela répondra à la question de savoir si le mécanisme fonctionne. Ensuite, regardez les registres et la mémoire (cf. exercice 3.19) et déterminez s'il y a un problème.*

Paramètres des sous-programmes

Lorsqu'on appelle un sous-programme, on espère normalement qu'il agira sur certaines données. Par exemple, dans le cas de la multiplication, on souhaite transmettre les deux nombres dont le

sous-programme va effectuer la multiplication. Nous avons vu dans le cas de la multiplication que ce sous-programme s'attendait à trouver le multiplicateur et le multiplicande dans des emplacements donnés de la mémoire.

Ceci illustre la première méthode de passer des paramètres : par la mémoire. On utilise deux autres techniques et les paramètres peuvent être transmis de trois manières :

1. par des registres,
2. par la mémoire,
3. par la pile.

— Des *registres* peuvent être utilisés pour passer des paramètres. C'est une solution avantageuse, pourvu que les registres soient disponibles, car on n'a pas besoin d'utiliser un emplacement-mémoire fixe. Le sous-programme reste indépendant de la mémoire. Si l'on fait appel à un emplacement-mémoire déterminé, tout autre utilisateur du sous-programme doit prendre un soin absolu d'utiliser la même convention et veiller à ce que l'emplacement-mémoire soit effectivement disponible (cf. l'exercice 3.20 ci-dessus). C'est pourquoi, dans beaucoup de cas, on réserve un bloc de cases-mémoires, simplement pour passer des paramètres entre les divers sous-programmes.

— *L'utilisation de la mémoire* a l'avantage d'une plus grande souplesse (on peut transmettre plus de données), mais conduit à des performances plus faibles et fait dépendre le sous-programme d'une zone mémoire déterminée.

— le dépôt des paramètres dans la *pile* a le même avantage que l'utilisation de registres : il est indépendant de la mémoire. Le sous-programme sait simplement qu'il est censé recevoir, disons deux paramètres, qui se trouvent au sommet de la pile. Naturellement il y a un inconvénient : cela remplit la pile de données, et par suite, diminue le nombre de niveaux d'appels de sous-programmes possibles.

Le choix est entre les mains du programmeur. Dans le cas général, on souhaite rester, le plus possible, indépendant d'adresses-mémoire déterminées.

Si l'on ne dispose pas de registres, la meilleure solution qui reste est généralement la pile. Néanmoins s'il y a à passer à un programme une grande quantité d'informations, elle devra résider en mémoire. Une manière élégante de résoudre le problème de passer un bloc de données est de transmettre simplement un pointeur vers l'information. Le *pointeur* est l'adresse de début du bloc. Un pointeur peut être transmis dans un registre (dans le cas du 6502, cela limite le pointeur à 8 bits) ou alors dans la pile (on peut utiliser deux cases de la pile pour stocker une adresse de 16 bits).

Enfin, si aucune de ces deux solutions n'est applicable, on peut dans le sous-programme faire la convention que la donnée sera à une adresse-mémoire fixée (la «boîte aux lettres»).

Exercice 3-21 : *Laquelle des trois méthodes ci-dessus est la meilleure pour la récursivité ?*

Bibliothèque de sous-programmes

Il y a un avantage important à structurer les portions d'un programme en un ensemble de sous-programmes identifiables : ils peuvent être mis au point indépendamment et avoir un nom qui rappelle leur fonction. S'ils sont utilisés dans d'autres parties du programme, ils deviennent partageables, et on peut ainsi construire une bibliothèque de sous-programmes utiles. Néanmoins il n'y a pas de panacée en programmation. Utiliser des sous-programmes systématiquement pour tout groupe d'instructions qui peuvent être réunies par fonction peut conduire à une mauvaise efficacité d'exécution. Le programmeur habile aura à peser les avantages vis-à-vis des inconvénients.

RECAPITULATION

Ce chapitre a présenté la façon dont les instructions manipulent l'information à l'intérieur du 6502. Des algorithmes de plus en plus complexes ont été introduits et traduits en programmes. Les principaux types d'instructions ont été présentés et utilisés.

Des structures importantes, telles que la boucle, la pile et les sous-programmes, ont été introduites et utilisées.

Vous devriez maintenant avoir acquis une compréhension des bases de la programmation et des principales techniques utilisées dans les applications standards. Etudions à présent les instructions disponibles.

4

LE JEU D'INSTRUCTIONS DU 6502

1re PARTIE-DESCRIPTION GENERALE

INTRODUCTION

Dans ce chapitre les diverses classes d'instructions qui doivent être disponibles dans un ordinateur d'usage général vont d'abord être analysées. Puis chacune des instructions du 6502, va être examinée en détail, ainsi que son utilisation, la façon dont elle affecte les indicateurs d'état et les divers modes d'adressage qu'on peut utiliser en conjonction avec elle. Une discussion détaillée des Techniques d'adressage sera présentée au Chapitre 5.

CLASSES D'INSTRUCTIONS

Les instructions peuvent être classées de nombreuses façons et il n'y a pas de classification standard. Nous distinguerons ici cinq catégories principales d'instructions.

1. Transfert de données.
2. Traitement de données.
3. Tests et branchements.
4. Entrées-sorties.
5. Contrôle.

Examinons maintenant tour à tour chacune de ces classes d'instructions.

Transfert de données

Les instructions de transfert de données transmettent une donnée de 8 bits entre deux registres ou entre un registre et la mémoire, ou entre un registre et un boîtier d'entrée-sortie. Il peut y avoir des instructions de transfert spécialisées pour des registres qui jouent un rôle particulier, par exemple une instruction d'empilage et de dépilage est généralement disponible pour des manipulations de pile efficaces. Ces deux instructions transfèrent en une seule opération un mot de données entre le sommet de la pile et l'accumulateur, tout en mettant à jour le pointeur de pile automatiquement.

Traitement de données

Les instructions de traitement de données se répartissent en quatre catégories générales :

— opérations arithmétiques (telles que « plus » ou « moins ») ;
— opérations logiques (telles que ET, OU, OU exclusif) ;
— opérations de déplacement de bits et de décalage (y compris décalage, rotation, échange) ;
— incrémentation et décrémentation.

Il faut noter que pour des traitements de données efficaces, il serait désirable de disposer d'instructions arithmétiques puissantes telles que la multiplication ou la division. Malheureusement, sur la plupart des microprocesseurs elles ne sont pas disponibles. Il serait aussi désirable d'avoir des instructions de décalage et de déplacement puissantes telles que le décalage de n bits ou l'échange de quartets (où on échange la moitié droite et la moitié gauche d'un octet). Elles sont généralement absentes sur la plupart des microprocesseurs.

Avant d'examiner les instructions propres au 6502, rappelons la différence entre un *décalage* et une *rotation.* Un décalage déplace le contenu d'un registre ou d'une case-mémoire, d'un bit vers la gauche ou vers la droite. Le bit qui sort du registre se retrouve dans le bit de retenue. Le bit qui entre de l'autre côté est un « 0 ».

Dans le cas d'une *rotation,* le bit sortant va toujours dans la retenue. Mais le bit qui entre est l'ancienne valeur du bit de retenue. Ceci correspond à une rotation de 9 bits. Il serait souvent désirable d'avoir une véritable rotation sur 8 bits où le bit entrant par un côté soit celui qui sort de l'autre côté. La plupart des microprocesseurs n'en disposent pas, en général. Enfin lorsqu'on décale vers la droite, il est commode d'avoir un autre mode de décalage appelé *extension du signe* ou « décalage arithmétique à droite ». Lorsqu'on effectue des opérations sur des nombres en complément à 2, particulièrement lorsqu'on implante des routines d'arithmétique en virgule flottante,

il est souvent nécessaire de décaler un nombre négatif vers la droite. Lorsqu'on décale à droite un nombre négatif en complément à 2, le bit qui doit entrer sur la gauche est un « 1 ». Le bit de signe doit être répété autant de fois qu'on fait de décalages successifs. Malheureusement ce type de décalage n'existe pas sur le 6502. Il existe sur d'autres microprocesseurs.

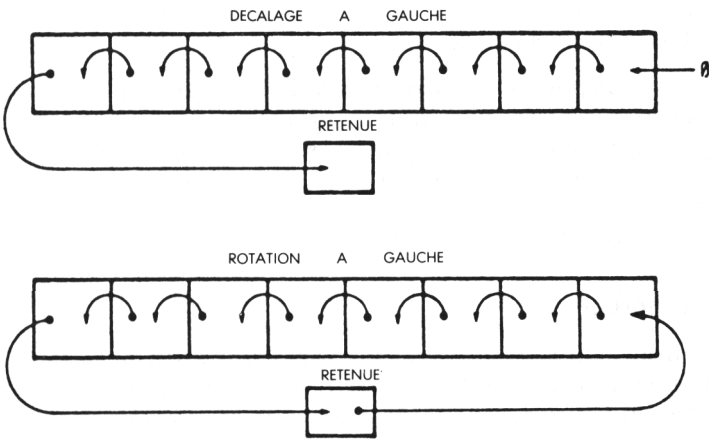


Fig. 4-1 : Décalage et rotation

Tests et branchements

Les instructions de test peuvent tester en général chacun des bits du registre d'état pour savoir s'ils valent « 0 » ou « 1 », ou des combinaisons de ceux-ci. Il est donc désirable d'avoir autant d'indicateurs que possible dans ce registre. En outre, il est commode de pouvoir tester des combinaisons de ces bits en une seule instruction. Enfin il est désirable de pouvoir tester *n'importe quel bit de n'importe quel registre* et de tester la valeur d'un registre comparée à celle d'un autre registre (inférieur, supérieur, égal).

Les instructions de test des microprocesseurs sont généralement limitées à des tests de bits isolés du registre d'état.

Les instructions de *saut* qui peuvent être disponibles se répartissent généralement en trois catégories :

— les sauts proprement dits, qui spécifient une adresse 16 bits complète ;

— les branchements qui se contentent souvent d'un déplacement sur 8 bits ;
— l'appel (« CALL ») qui est utilisé pour les sous-programmes.

Il est commode d'avoir des branchements à deux ou même trois voies, dépendant par exemple du résultat d'une comparaison : « plus grand », « plus petit », ou « égal ». Il est aussi commode d'avoir des opérations de saut où l'on saute en avant ou en arrière, de quelques instructions. Enfin, dans la plupart des boucles, il y a généralement à la fin, une opération d'incrémentation ou de décrémentation suivie d'un test et branchement.

Donc la disponibilité d'une instruction qui regroupe incrémentation/décrémentation plus test et branchement facilite nettement l'implantation efficace des boucles. Ceci n'est pas disponible sur la plupart des microprocesseurs. On ne dispose que de branchements simples combinés avec des tests simples. Naturellement, cela complique la programmation et nuit à l'efficacité.

Entrées-sorties

Les instructions d'entrées-sorties sont des instructions spécialisées pour la gestion des unités d'entrées-sorties. En pratique, presque tous les microprocesseurs utilisent des *entrées-sorties projetées en mémoire*. Cela signifie que les circuits d'entrées-sorties sont connectés au bus adresse exactement comme les boîtiers-mémoire et adressés comme eux. Ils sont vus par le programmeur comme des emplacements-mémoire. On peut donc appliquer toutes les opérations-mémoire aux organes voulus. Ceci a l'avantage de permettre d'appliquer une variété étendue d'instructions. L'inconvénient est que les instructions-mémoire exigent normalement 3 octets et sont, par suite, plus lentes. Pour une gestion efficace des entrées-sorties dans un tel environnement, il est désirable de disposer d'un mécanisme d'adressage court de sorte que les circuits d'E/S dont la vitesse de gestion est cruciale puissent résider en page 0. Cependant si l'adressage page 0 est disponible, il est d'habitude utilisé pour de la mémoire RAM ce qui empêche de l'utiliser effectivement pour les entrées-sorties.

Instructions de contrôle ou de commande

Les instructions de contrôle fournissent des signaux de synchronisation et elles peuvent suspendre ou interrompre un programme. Elles peuvent aussi fonctionner comme un break ou interruption simulée. (Les interruptions sont décrites au Chapitre 6 sur les techniques d'entrées-sorties).

INSTRUCTIONS DISPONIBLES SUR LE 6502

Instructions de transfert de données

Le 6502 a un jeu complet d'instructions de transfert de données, sauf pour le chargement du pointeur de pile qui est peu souple. Le contenu de l'accumulateur peut être échangé avec une case mémoire par les instructions LDA (charger) et STA (ranger). Il en est de même des registres X et Y avec, respectivement, les instructions LDX, LDY et STX, STY. Il n'y a pas de chargement direct de S. Il y a naturellement des transferts entre registres : ce sont TAX (transférer A dans X), TAY, TSX, TXA, TXS, TYA. Il y a une légère dissymétrie du fait que le pointeur de pile peut être échangé avec X mais pas avec Y.

Il n'y a pas d'opérations entre mémoire et mémoire avec 2 adresses comme « ajouter les contenus de LOC1 et LOC2 ».

Opérations avec la pile

Deux instructions « d'empilage » et de « dépilage » sont disponibles. Elles transfèrent l'accumulateur A ou le registre d'état (P) vers le sommet de la pile en mémoire tout en mettant à jour le pointeur de pile S. Ce sont PHA et PHP. Les opérations inverses sont PLA et PLP (dépiler A et dépiler P) qui transfèrent le sommet de pile respectivement vers A ou vers P.

Traitement de données

Arithmétique

L'ensemble restreint habituel de fonctions arithmétiques, logiques ou de décalage est disponible. Les opérations arithmétiques sont : ADC et SBC. ADC est une addition avec retenue, et il n'y a pas d'addition sans retenue sur le 6502. C'est un inconvénient mineur car il faut exécuter une instruction CLC avant toute addition. La soustraction est effectuée par SBC. Il existe un mode décimal spécial qui autorise l'addition et la soustraction directes des nombres exprimés en DCB. Dans beaucoup d'autres microprocesseurs, seule l'une de ces instructions DCB est disponible (soit « plus », soit « moins ») et nécessite un code opératoire distinct. La présence de l'indicateur de mode décimal multiplie par deux le nombre d'instructions arithmétiques effectivement disponibles.

Incrémentation/Décrémentation

Les opérations d'incrémentation et de décrémentation existent sur la mémoire et sur les registres index X et Y, mais pas sur l'accumulateur.

Ce sont respectivement INC et DEC qui agissent sur la mémoire, INX, INY et DEX, DEY qui opèrent sur les registres index X et Y.

Opérations logiques

Les opérations logiques sont les opérations classiques ET, OU et OU exclusif. Le rôle de chacune va être décrit.

AND (ET)

Chaque opération logique est caractérisée par une table de vérité qui fournit la valeur logique du résultat en fonction des entrées. La table de vérité pour le ET apparaît ci-dessous :

0 ET 0 = 0
0 ET 1 = 0
1 ET 0 = 0
1 ET 1 = 1

L'opération « ET » (« AND ») est caractérisée par le fait que la sortie est « 1 » seulement si les deux entrées valent « 1 ». En d'autres termes, si l'une des entrées vaut « 0 », il est garanti que le résultat vaut « 0 ». Ce fait est utilisé pour mettre à zéro un bit dans un mot. Cela s'appelle un « masquage ».

Une des utilisations importantes de l'instruction AND est d'annuler ou masquer sélectivement un ou plusieurs bits dans un mot. Supposons par exemple que nous voulions annuler les quatre bits de droite dans un mot. Cela sera accompli par le programme suivant :

```
LDA   # MOT          MOT contient « 10101010 »
AND   # %11110000    LE MASQUE EST « 11110000 »
```

Supposons que MOT vaille « 10101010 ». Le résultat du programme est de laisser dans l'accumulateur la valeur « 10100000 ». Le symbole « % » sert à indiquer un nombre binaire.

Exercice 4.1 : *Ecrire un programme de trois lignes qui annule les bits 1 et 6 de MOT.*

Exercice 4.2 : *Que produit un masque égal à « 11111111 » ?*

ORA (OU INCLUSIF)

Cette instruction réalise l'opération OU inclusif. Elle est caractérisée par la table de vérité suivante :

0 OU 0 = 0
0 OU 1 = 1
1 OU 0 = 1
1 OU 1 = 1

Le OU logique est caractérisé par le fait que si l'un des opérandes est à 1, alors le résultat est toujours « 1 ». L'utilité évidente du « OU » est de mettre à « 1 » un bit quelconque dans un mot. Mettons à « 1 » les quatre bits les plus à droite de « MOT ». Le programme est :

```
LDA # MOT
ORA # %00001111
```

Si nous supposons que MOT contenait « 10101010 », la valeur finale présente dans l'accumulateur sera « 10101111 ».

Exercice 4.3 : *Qu'arriverait-il si on effectuait un ORA =%10101111 ?*

Exercice 4.4 : *Quel est l'effet d'un OU avec « FF » en hexadécimal ?*

EOR (OU EXCLUSIF)

EOR effectue le « OU exclusif ». Le OU exclusif diffère du OU inclusif que nous venons de décrire par un aspect : le résultat n'est « 1 » que si un seul des opérandes est égal à « 1 ». Si les deux opérandes sont égaux à « 1 », le « OU » normal a pour résultat « 1 ». Le OU exclusif donne pour résultat « O ».

La table de vérité est :

0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0

Le OU exclusif est utilisé pour les comparaisons. Si un bit est différent, le OU exclusif des deux mots sera non nul. En outre, dans le cas du 6502, on utilise le OU exclusif pour *complémenter* un mot, étant donné qu'il n'existe pas d'instruction spécifique pour cela. Ceci est accompli en effectuant le EOR d'un mot ne contenant que des uns. Le programme apparaît ci-dessous :

```
LDA # MOT
EOR # %11111111
```

Si nous supposons que MOT contenait « 10101010 », la valeur finale de l'accumulateur sera « 01010101 ». Nous pouvons vérifier que c'est le complément de la valeur de départ.

Exercice 4.5 : *Quel est l'effet de EOR # $00 ?*

Décalages

Le 6502 possède un décalage à gauche appelé ASL (arithmetic shift left = décalage arithmétique à gauche), et un décalage à droite

appelé LSR (logical shift right = décalage logique à droite). Ils seront détaillés ci-dessous. Le 6502 possède deux instructions de rotation : à gauche (ROL) et à droite (ROR).

Attention : les versions anciennes du 6502 ne possédaient que ROL. S'assurer du modèle livré auprès du constructeur.

Comparaisons

Les registres X, Y et A peuvent être comparés au contenu d'une adresse mémoire à l'aide des instructions CPX, CPY, et CMP.

Tests et branchements

Comme les tests portent presque exclusivement sur les indicateurs du registre P, examinons quels indicateurs sont disponibles dans le 6502. Le contenu du registre d'état apparaît Figure 4-2 ci-dessous.

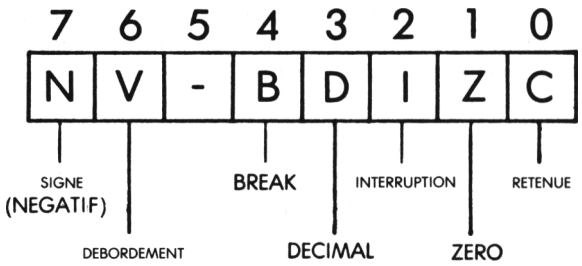


Fig. 4-2 : Le registre d'état

Examinons les fonctions des indicateurs, de gauche à droite.

Signe

Le bit le plus à gauche est le bit de signe ou bit « négatif » ; quand N est à « 1 », il indique que la valeur d'un résultat est négative dans la notation en complément à 2. En pratique, l'indicateur N est identique au bit 7 du résultat qu'on vient d'obtenir. Il est positionné par toutes les instructions de transfert et de traitement de données.

L'indicateur N est, dans la plupart des cas, identique au bit 7 de l'accumulateur. Il en résulte que le bit 7 de l'accumulateur est le seul bit qui peut être testé commodément avec une seule instruction. Pour tester tout autre bit de l'accumulateur, il est nécessaire de faire des décalages. Dans tous les cas où on veut tester rapidement un

mot, la position la plus favorable sera donc le bit 7. C'est pourquoi les indicateurs d'état des entrées-sorties sont normalement connectés à la ligne 7 du bus de données ; lorsqu'on lit l'état d'un circuit d'entrée-sortie, on lit simplement dans l'accumulateur le contenu du registre d'état puis on teste le bit N.

Le second bit de l'accumulateur qui est le plus facile à tester est le bit 0. Cependant il nécessite un décalage à droite d'un cran qui le transfère dans le bit de retenue, où il peut alors être testé.

Le bit N ne peut être positionné par le programmeur. Il est positionné automatiquement par des instructions.

Les instructions qui affectent N sont : ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, TAX, TAY, TSX, TXA, TYA.

Enfin, le bit 6 peut également être testé commodément grâce à l'instruction BIT, décrite plus loin.

Débordement

Le rôle de l'indicateur de débordement (overflow) a déjà été présenté au Chapitre 3, dans la section relative aux opérations arithmétiques. Il sert à indiquer que le résultat d'une addition ou d'une soustraction en complément à 2 risque d'être incorrect à cause d'un débordement du bit 6 vers le bit 7, c'est-à-dire, vers le bit de signe. On doit faire appel à une routine spéciale de correction à chaque fois que ce bit est positionné. Si on n'est pas en complément à 2, mais en représentation binaire directe, le bit de débordement est équivalent à une retenue du bit 6 vers le bit 7.

L'instruction BIT utilise ce bit de façon particulière. Un des effets de cette instruction est de copier dans le bit « V » le bit 6 de la donnée qu'on est en train de tester. Le bit V est positionné par les instructions ADC, BIT, CLV, PLP, RTI et SBC.

Break

L'indicateur de « break » est positionné automatiquement par le processeur, si on crée une interruption par la commande BRK. Il permet de faire la différence entre une interruption programmée et une interruption hardware. Aucune autre instruction-utilisateur ne peut le modifier.

Décimal

L'utilisation de cet indicateur a déjà été discutée au Chapitre 3 dans la section sur les programmes arithmétiques. Quand D est mis

à « 1 », le processeur opère en *mode DCB* et quand il est à « 0 », le processeur fonctionne en *mode binaire.* Cet indicateur est positionné par quatre instructions : CLD, PLP, RTI, SED.

Inhibition d'interruption

Ce bit de masque des interruptions peut être positionné explicitement par le programmeur à l'aide des instructions CLI, SEI et PLP, ou par le micro-processeur lors du « reset », ou pendant une interruption. Son effet est d'inhiber toute interruption subséquente.

Les instructions qui positionnent ce bit sont : BRK, CLI, PLP, RTI et SEI.

Zéro

L'indicateur Z signale, quand il est égal à « 1 », que le résultat d'un transfert ou d'une opération est zéro. Il est aussi positionné par l'instruction de comparaison. Il n'y a pas d'instruction spécifique pour forcer le bit Z à « 0 » ou à « 1 ». Mais on peut obtenir facilement le même effet. Pour annuler le bit Z on peut, par exemple, exécuter l'instruction suivante : LDA #10.

Le bit Z peut être positionné par de nombreuses instructions : ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TAY, TSX, TXA, TYA.

Retenue

On a vu que le bit de retenue a un double usage. Sa première utilisation est d'indiquer s'il y a eu une retenue ou un emprunt lors d'opérations arithmétiques.

La seconde utilisation est de recevoir le bit sortant d'un registre lors d'une opération de décalage ou de rotation. Les deux rôles ne sont pas nécessairement confondus dans un ordinateur, et ils ne le sont pas sur les gros ordinateurs. Mais cette approche gagne du temps dans le cas d'un microprocesseur, en particulier pour la réalisation de la multiplication ou de la division. Le bit de retenue peut être forcé à « 0 » ou à « 1 » explicitement par des instructions spécialisées : CLC, SEC.

Les instructions qui positionnent le bit de retenue sont : ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP, ROL, ROR, RTI, SBC, SEC.

Instructions de tests et branchements

Il est possible sur le 6502 de tester les bits du registre d'état par rapport à « 0 » ou à « 1 ». Il y a 4 bits testables, et, par suite, 8 instructions de branchements différentes ; ce sont :

— BMI (branch on minus = branchement si moins), et BPL (branch on plus = branchement si plus), qui, naturellement, testent le bit N.

— BCC (branch on carry clear = branchement si retenue à « 0 ») et BCS (branch on carry set = branchement si retenue à « 1 ») qui testent C.

— BEQ (branchement si résultat nul) et BNE (branchement si résultat non nul) qui testent Z.

— BVS (branch on overflow set = branchement si V = « 1 ») et BVC (branch on overflow clear = branchement si V = « 0 ») qui testent V.

Le test et le branchement sont effectués au cours de la même instruction. Tous les branchements spécifient un déplacement relatif à l'instruction en cours. Comme le champ affecté au déplacement est de 8 bits cela permet un déplacement compris entre — 128 et + 127 (exprimé en complément à deux). Le déplacement est ajouté à l'adresse de la première instruction qui suit le branchement. Comme tous les branchements font deux octets de long, il en résulte un déplacement effectif compris entre — 128 + 2 = — 126 et 127 + 2 = + 129.

Il y a en plus deux instructions de saut inconditionnel : JMP et JSR, JMP est un saut vers une adresse de 16 bits. JSR est l'appel d'un sous-programme. Il fait sauter à une nouvelle adresse en préservant automatiquement le compteur ordinal dans la pile. Comme elles sont inconditionnelles, ces deux instructions de saut sont généralement précédées par une instruction de « test et branchement ».

Il existe deux instructions de retour : RTI, retour d'interruption, qui sera discutée dans la section sur les interruptions, et RTS, retour de sous-programme, qui récupère l'adresse de retour dans la pile, (et incrémente le SP). Il existe deux instructions particulières servant spécialement pour des tests de bits et des comparaisons : BIT et CMP.

L'instruction BIT effectue un « ET » entre un emplacement mémoire et l'accumulateur. Un point important est qu'*elle ne modifie pas le contenu de l'accumulateur.* L'indicateur N est mis à la valeur du bit 7 de la case-mémoire testée, tandis que V est mis à la valeur du bit 6. Enfin le bit Z indique le résultat du ET. Z est mis à « 1 » si le résultat est « 0 ». Généralement, on charge un masque dans

l'accumulateur et on teste successivement différentes cases-mémoire à l'aide d'instructions BIT. Si le masque contient un seul « 1 », par exemple, cela permet de tester si un mot quelconque de la mémoire a un « 1 » dans cette même position. En pratique, on n'a besoin de masque que pour tester les positions 0 à 5 puisque les positions 6 et 7 sont automatiquement copiées respectivement dans les indicateurs V et N. Elles n'ont donc pas besoin du masque.

L'instruction CMP compare le contenu de la case-mémoire indiquée à l'accumulateur en le soustrayant de l'accumulateur. Le résultat de la comparaison sera donc indiqué par l'état des bits Z et C. On peut détecter l'égalité, la supériorité et l'infériorité. La valeur de l'accumulateur n'est pas modifiée par la comparaison. CPX et CPY comparent respectivement avec X et avec Y.

Instructions d'entrées-sorties

Il n'y a pas d'instructions d'entrées-sorties spécialisées sur le 6502. Les entrées-sorties sont traitées comme des adresses-mémoire.

Instructions de contrôle

Les instructions de contrôle comprennent des instructions spécialisées pour mettre les indicateurs à « 0 » ou à « 1 ». Ce sont CLC, CLD, CLI, CLV, qui annulent respectivement les bits C, D, I et V et SEC, SED, SEI qui mettent à « 1 » C, D et I. Les autres instructions de contrôle sont BRK et NOP.

L'instruction BRK est équivalente à une interruption software ; elle sera décrite au Chapitre 7 dans la section sur les interruptions.

L'instruction NOP n'a aucun effet. Elle est habituellement utilisée pour allonger une boucle. Enfin deux broches spéciales du 6502 déclenchent un mécanisme d'interruption : ceci sera expliqué au Chapitre 6 sur les techniques d'entrées-sorties. C'est une possibilité de contrôle hardware (broches IRQ et NMI).

Examinons maintenant chaque instruction en détail.

2e PARTIE-LES INSTRUCTIONS

ABREVIATIONS

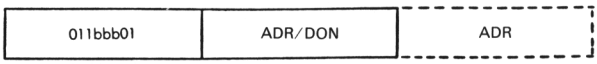
A	accumulateur
M	adresse-mémoire spécifiée
P	registre d'état
S	pointeur de pile
X	registre index
Y	registre index
DON	donnée spécifiée
HEX	hexadécimal
PC	compteur ordinal
PCH	partie haute de PC
PCL	partie basse de PC
PILE	contenu du sommet de la pile
V	ou logique
Λ	et logique
∀	ou exclusif
●	changement
←	« prend la valeur » (assignation)
( )	contenu de

ADC — Addition avec retenue

Fonction :

A ← (A) + DONNEE + C

Format :



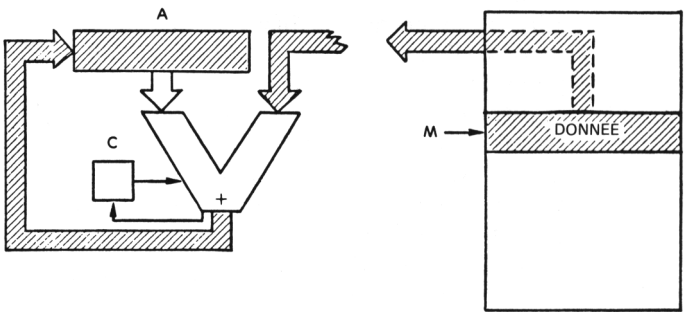
Description :

Ajouter à l'accumulateur le contenu de l'adresse-mémoire ou du littéral plus le bit de retenue. Le résultat est laissé dans l'accumulateur.

Remarques :

— ADC peut opérer soit en mode décimal, soit en mode binaire : les indicateurs doivent être positionnés à la valeur correcte ;
— pour ajouter sans retenue, il faut annuler l'indicateur C (CLC).

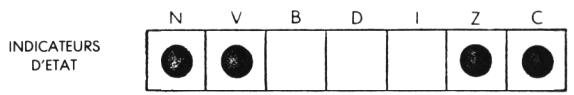
Circuit des informations :



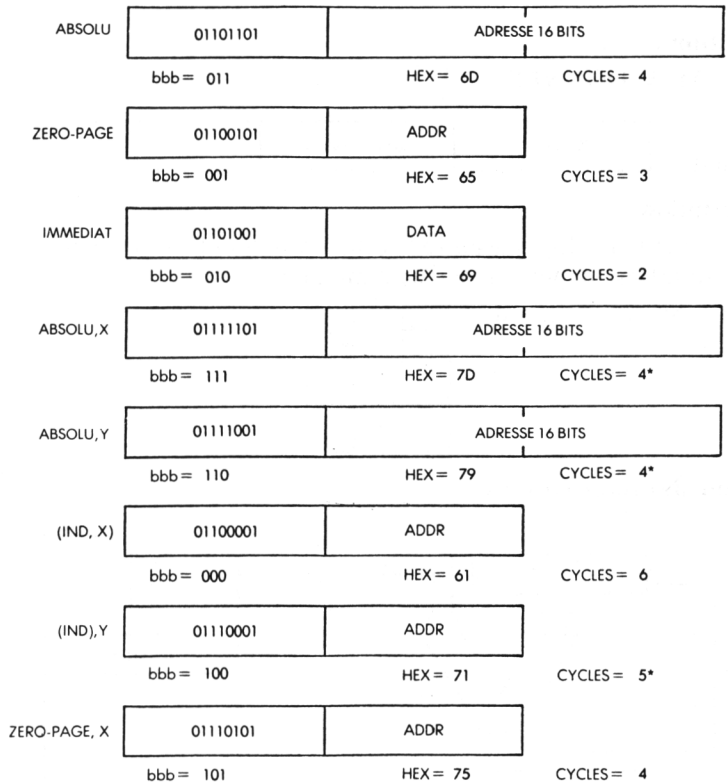
Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0-PAGE	IMMEDIAT	ABS. X	ABS. Y	(IND. X)	(IND). Y	0-PAGE. X	0-PAGE. Y	RELATIF	INDIRECT
HEX			6D	65	69	7D	79	61	71	75			
OCTETS			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

*PLUS 1 CYCLE SI UNE LIMITE DE PAGE EST FRANCHIE



Codes opération :



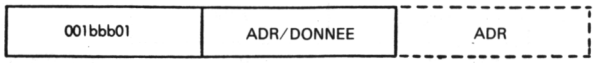
AND

ET logique (AND)

Fonction :

A ← (A) ∧ DONNEE

Format :



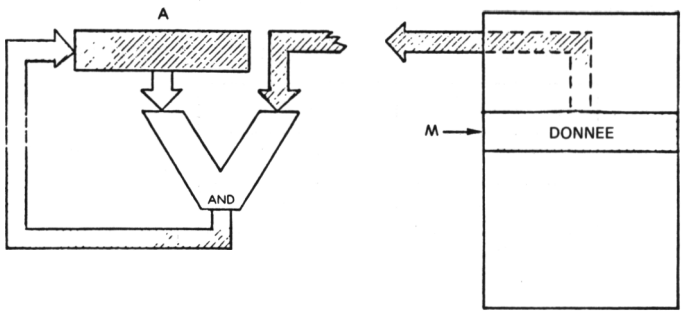
Description :

Effectue le ET logique entre l'accumulateur et la donnée indiquée. Le résultat est déposé dans l'accumulateur.

La table de vérité est

A∧M	0	1
0	0	0
1	0	1

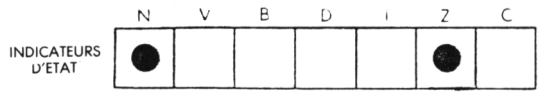
Circuit des informations



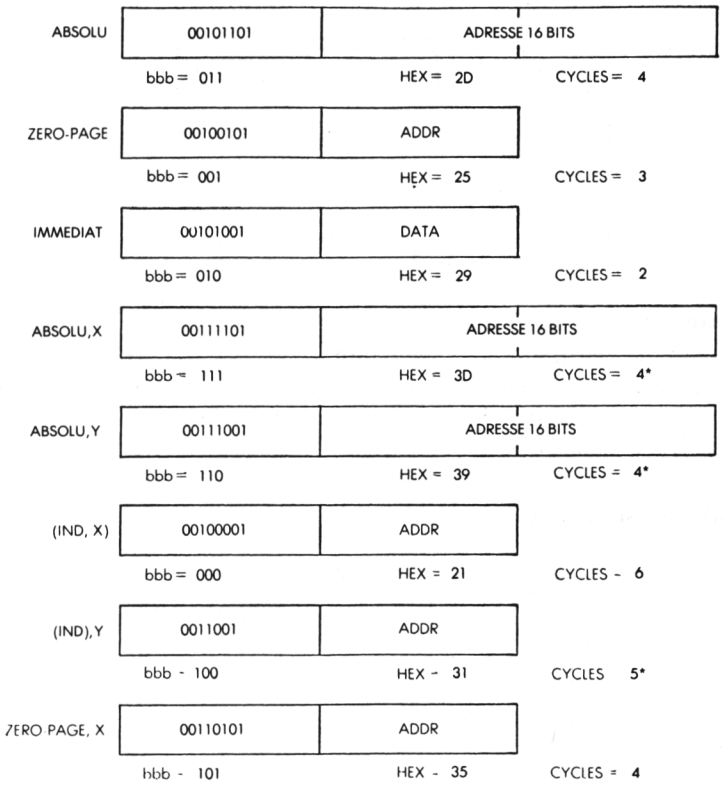
Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0-PAGE	IMMEDIAT	ABS. X	ABS. Y	(IND. X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIF	INDIRECT
HEX			2D	25	29	3D	39	21	31	35			
OCTETS			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

*PLUS 1 CYCLE SI UNE LIMITE DE PAGE EST FRANCHIE

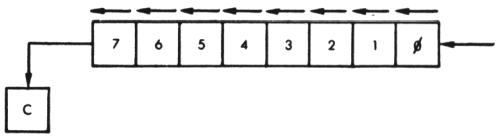


Codes opération :



ASL — Décalage arithmétique gauche

Fonction :

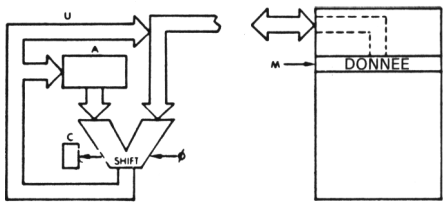


Format :

Description :

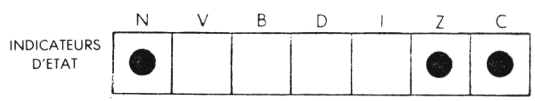
Décale le contenu de l'accumulateur ou de l'emplacement mémoire d'un bit vers la gauche. Il entre un O sur la droite. Le bit 7 tombe dans la retenue. Le résultat est déposé dans la source : accumulateur ou mémoire.

Circuit des informations

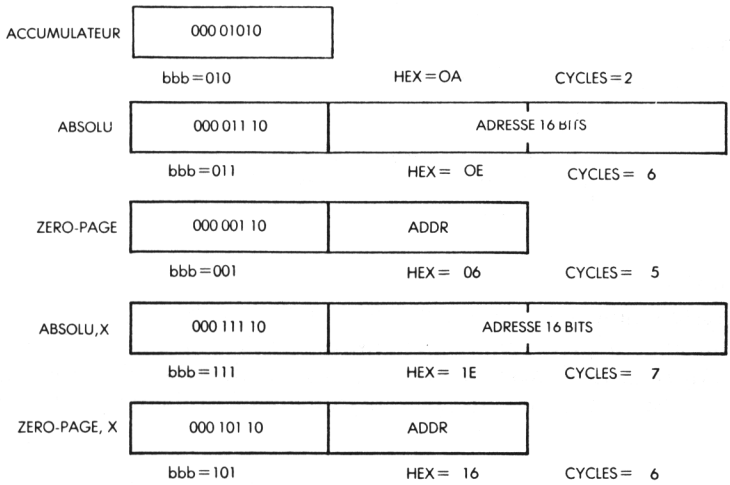


Modes d'adressage

	IMPLICITE	ACCUMU-LATEUR	ABSOLU	0-PAGE	IMMEDIAT	ABS. X	ABS. Y	(IND. X)	(IND). Y	0-PAGE. X	0-PAGE. Y	RELATIF	INDIRECT
HEX		ØA	ØE	Ø6		1E				16			
OCTETS		1	3	2		3				2			
CYCLES		2	6	5		7				6			
bbb		010	011	001		111				101			



Codes opération :

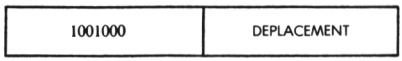


BCC — Branchement si pas de retenue

Fonction :

Va à l'adresse indiquée si C = 0.

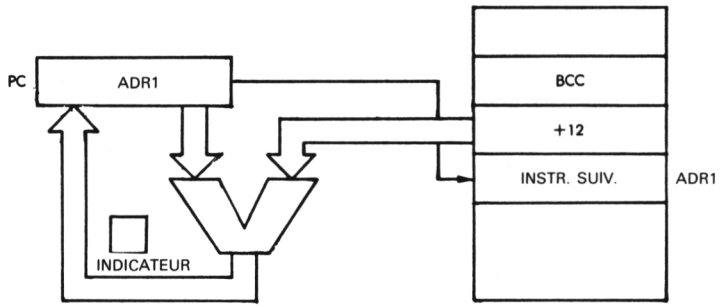
Format :



Description :

On teste l'indicateur de retenue. Si C = 0 on se branche à adresse courante plus déplacement signé (entre — 128 et + 127). Si C = 1, on ne fait rien. Le déplacement est ajouté à l'adresse de l'instruction suivant le BCC. Il en résulte un déplacement effectif entre — 126 et + 129.

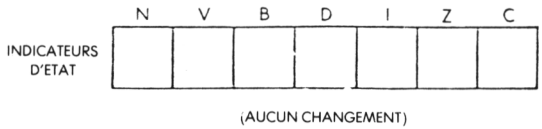
Circuit des informations :



Mode d'adressage :

Uniquement relatif.

HEX = 90, octets = 2, cycles = 2 + 1 si le branchement a lieu
+ 2 si on change de page.



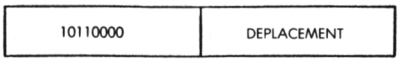
BCS

Branchement si retenue

Fonction :

Va à l'adresse indiquée si C = 1.

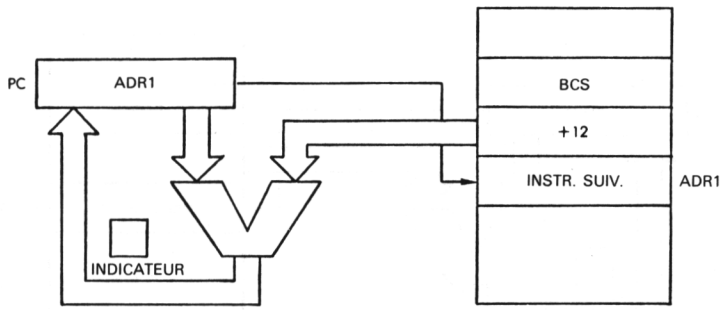
Format :



Description :

On teste l'indicateur de retenue. Si C = 1, on se branche à adresse courante plus le déplacement signé (de — 128 à + 127). Si C = 0, on ne fait rien. Le déplacement est ajouté à l'adresse de l'instruction qui suit le BCS. Il en résulte un déplacement effectif de — 126 à + 129.

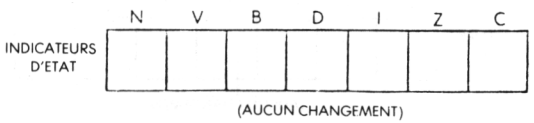
Circuit des informations



Mode d'adressage :

Uniquement relatif.

HEX = B0, octets = 2, cycles = 2 + 1 si le branchement a lieu
+ 2 si on change de page

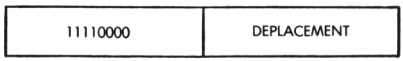


BEQ — Branchement si égalité

Fonction :

Va à l'adresse indiquée si Z = 1 (c'est-à-dire résultat = 0).

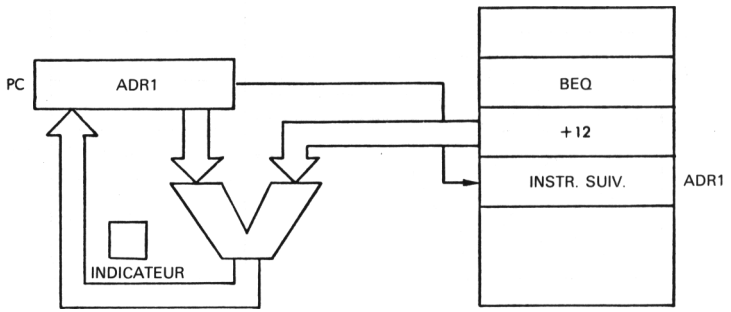
Format :



Description :

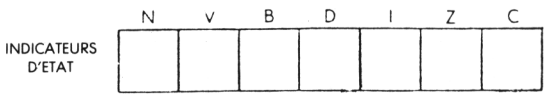
On teste l'indicateur Z. Si Z = 1, on se branche à adresse courante + déplacement signé (de — 128 à + 127). Si Z = 0, on ne fait rien c'est-à-dire qu'on passe à l'instruction suivante. Le déplacement est ajouté à l'adresse de l'instruction qui suit le BEQ. Il en résulte un déplacement effectif de — 126 à + 129.

Circuit des informations



Mode d'adressage :

Uniquement relatif
HEX = F0, octets = 2, cycles = 2 + 1 si le branchement a lieu
+ 2 si on change de page



(AUCUN CHANGEMENT)

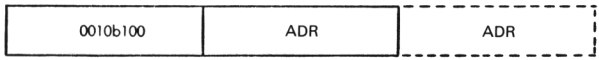
BIT

Test de bits en mémoire

Fonction :

$Z \leftarrow \overline{(A) \wedge (M)}$, $N \leftarrow (M^7)$, $V \leftarrow (M^6)$

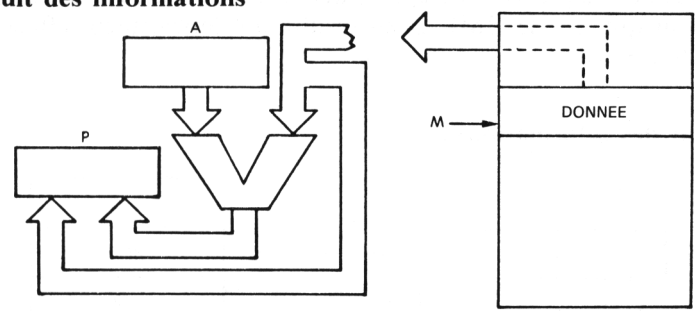
Format :



Description :

On effectue le ET entre A et M mais le résultat n'est pas déposé dans A. Le résultat est indiqué par Z : Z = 1 s'il y a accord, 0 sinon. En outre, les bits 6 et 7 de la mémoire sont transférés dans les bits V et N du registre d'état. Le contenu de A reste inchangé.

Circuit des informations

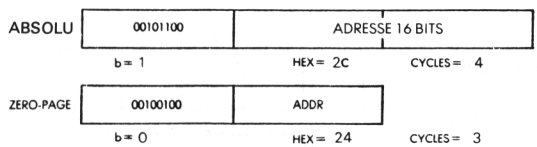


Modes d'adressage

	IMPLICITE	ACCUMULATEUR	ABSOLUT	0-PAGE	IMMEDIAT	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIF	INDIRECT
HEX			2C	24									
OCTETS			3	2									
CYCLES			4	3									
bbb			011	001									

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT	M^7	M^6				●	

Codes opération :

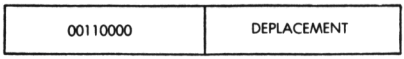


BMI — Branchement à moins

Fonction :

Va à l'adresse indiquée si N = 1 (résultat < 0).

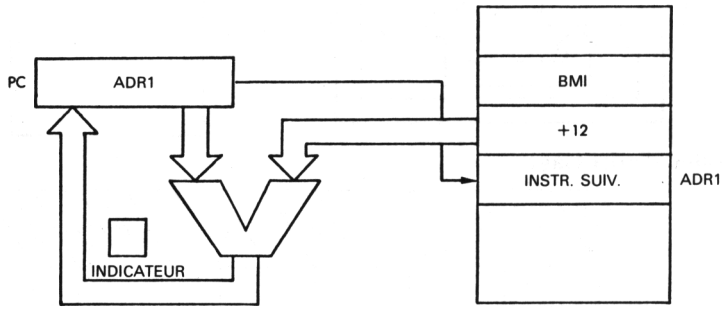
Format :



Description :

On teste l'indicateur N (signe). Si N = 1, on se branche à l'adresse courante plus le déplacement signé (de — 128 à + 127). Si N = 0, on continue en séquence. Le déplacement est ajouté à l'adresse de l'instruction qui suit le BMI. Il en résulte un déplacement effectif de — 126 à + 129.

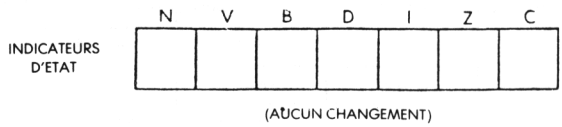
Circuit des informations :



Mode d'adressage :

Uniquement relatif.

HEX = 30, octets = 2, cycles = 2 + 1 si le branchement a lieu
+ 2 si on change de page

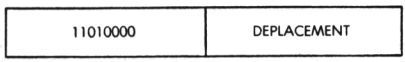


BNE — Branchement si non égal à zéro

Fonction :

Va à l'adresse indiquée si Z = 0 (résultat ≠0).

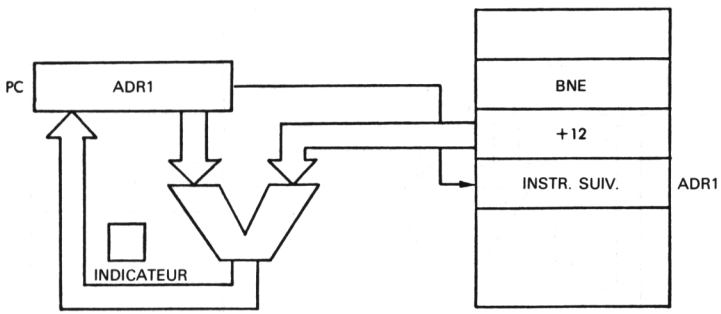
Format :



Description :

Teste le bit Z. Si le résultat n'est pas égal à 0 (Z = 0), on se branche à adresse courante + déplacement signé (de — 128 à + 127). Si Z = 1, on continue en séquence. Le déplacement est ajouté à l'adresse de l'instruction qui suit le BNE. Il en résulte un déplacement effectif de — 126 à + 129.

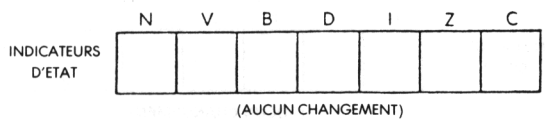
Circuit des informations :



Mode d'adressage :

Uniquement relatif

HEX = D0, octets = 2, cycles = 2 + si le branchement a lieu
+ 2 si on change de page

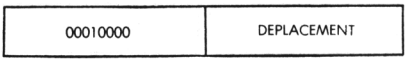


BPL — Branchement si plus

Fonction :

Va à l'adresse indiquée si N = 0 (Résultat ⩾ 0)

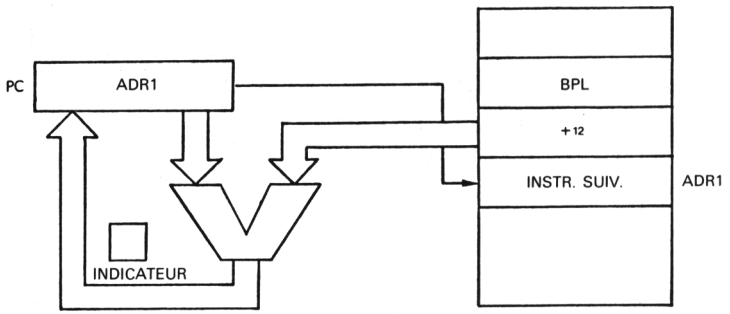
Format :



Description :

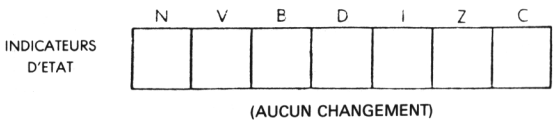
On teste l'indicateur N (signe). Si N = 0 (résultat positif), on se branche à adresse courante plus déplacement signé (de — 128 à + 127). Si N = 1, on continue en séquence. Le déplacement est ajouté à l'adresse de l'instruction qui suit le BPL. Il en résulte un déplacement effectif de — 126 à + 129.

Circuit des informations :



Mode d'adressage :

Uniquement relatif
HEX = 10, OCTETS = 2, cycles = 2 + 1 si le branchement a lieu
+ 2 si on change de page



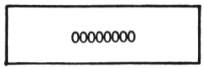
BRK

Interruption software

Fonction :

EMPILE (PC) + 2 PUIS (P), PC ← (FFFE,FFFF)

Format :

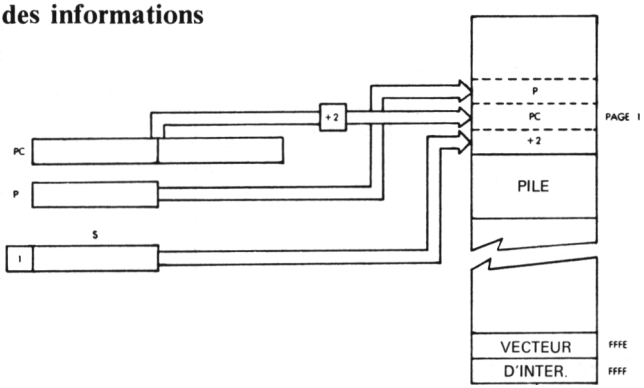


Description :

Fonctionne comme une interruption : le compteur ordinal est empilé, ainsi que le registre d'état P. Le contenu respectif des cases-mémoire FFFE et FFFF et envoyé dans PCL et PCH. La valeur de P empilée a l'indicateur B mis à 1 pour différencier BRK et IRQ.

Important : à la différence d'une interruption, c'est PC + 2 qui est sauvé. Cela peut ne pas être l'adresse de la prochaine instruction et une correction peut s'avérer nécessaire. Il est dû à l'usage du BRK qui consiste à remplacer dans un programme existant une instruction qui a le plus souvent deux octets.

Circuit des informations



Mode d'adressage :

Uniquement implicite

HEX = 00, octet = 1, cycles = 7

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT			★		1		

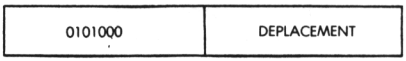
* B est à 1 dans la pile

BVC Branchement si pas de débordement

Fonction :

Va à l'adresse indiquée si V = 0

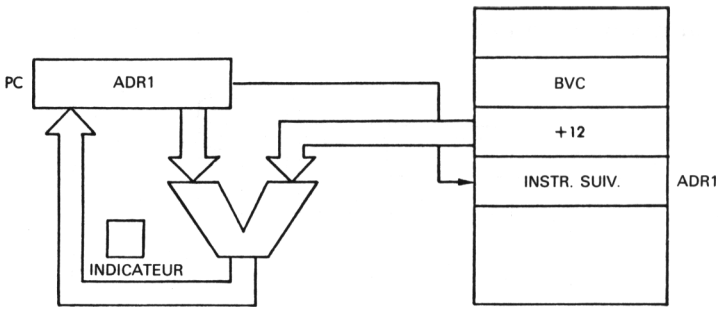
Format :



Description :

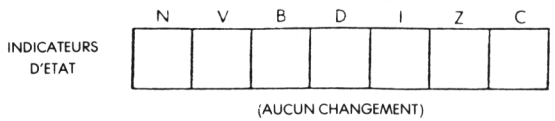
On teste l'indicateur de débordement (V). Si V = 0 (pas de débordement), on se branche à adresse courante plus déplacement signé (de — 128 à + 127). Si V = 1 on continue en séquence. Le déplacement est ajouté à l'adresse de l'instruction qui suit le BVC. Il en résulte un déplacement effectif de — 126 à + 129.

Circuit des informations :



Mode d'adressage :

Uniquement relatif
HEX = 50, octets = 2, cycles = 2 + 1 si le branchement a lieu
+ 2 si on change de page

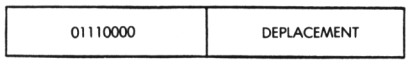


BVS — Branchement si débordement

Fonction :

Va à l'adresse indiquée si V = 1

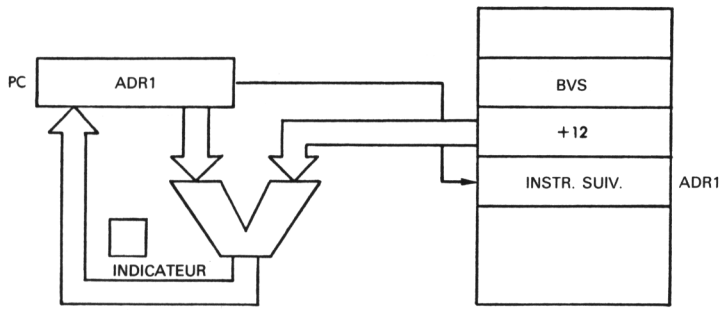
Format :



Description :

On teste l'indicateur de débordement (V). S'il y a eu un débordement (V = 1), on se branche à adresse courante plus déplacement signé (de — 128 à + 127). Si V = 0, on continue en séquence. Le déplacement est ajouté à l'adresse de l'instruction qui suit le BVS. Il en résulte un déplacement effectif de — 126 à + 129.

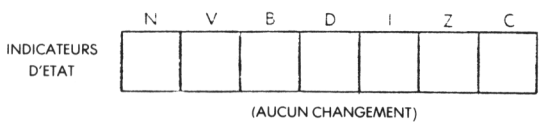
Circuit des informations :



Mode d'adressage :

Uniquement relatif

HEX = 70, octets = 2, cycles = 2 + 1 si le branchement a lieu
+ 2 si on change de page



CLC — Mise à zéro de la retenue

Fonction :

$C \leftarrow \emptyset$

Format :

00011000

Description :

On met à 0 le bit de retenue. C'est souvent nécessaire avant un ADC.

Mode d'adressage :

Uniquement implicite

HEX = 18, octet = 1, cycles = 2

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT							∅

CLD — Mise à zéro du mode décimal

Fonction :

$D \leftarrow \emptyset$

Format :

11011000

Description :

L'indicateur D est mis à 0 ce qui établit le mode binaire pour ADC et SBC.

Mode d'adressage :

Uniquement implicite

HEX = D8, octet = 1, cycles = 2

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT				∅			

CLI Mise à zéro du masque des interruptions

Fonction :

$1 \leftarrow \emptyset$

Format :

01011000

Description :

On met à 0 le masque des interruptions. Ceci autorise les interruptions. Une routine de traitement d'interruption doit toujours mettre I à 0 sinon d'autres interruptions risquent d'être perdues.

Mode d'adressage :

Uniquement implicite

HEX = 58, octet = 1, cycles = 2

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT					$\emptyset$		

CLV Mise à zéro de l'indicateur de débordement

Fonction :

$V \leftarrow \emptyset$

Format :

10111000

Description :

L'indicateur de débordement est mis à zéro.

Mode d'adressage :

Uniquement implicite

HEX = B8, octet = 1, cycles = 2

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT		Ø					

CMP — Comparer à l'accumulateur

Fonction :

(A) — DONNEE → NZC :

+ (A > DONNEE)	=	— (A<DONNEE)
-01	011	-00

(EN BINAIRE ABSOLU)

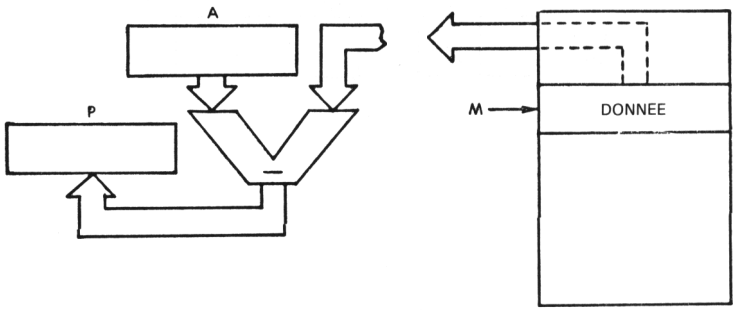
Format :

110bbb01	ADR/DONNEE	ADR

Description :

La donnée indiquée est soustraite de A. Le résultat n'est pas rangé dans A, mais les indicateurs N, Z et C sont positionnés selon que le résultat est positif, nul ou négatif. La valeur de l'accumulateur n'est pas modifiée. Z est positionné par une égalité : mis à zéro autrement. N est conditionné par le bit de signe C est positionné quand (A) > donnée. CMP est généralement suivi d'un branchement : BCC détecte (A) < donnée. BEQ détecte (A) = donnée. BCS détecte (A) ⩾ donnée et BEQ suivi de BCS détecte (A) > donnée.

Circuit des informations :



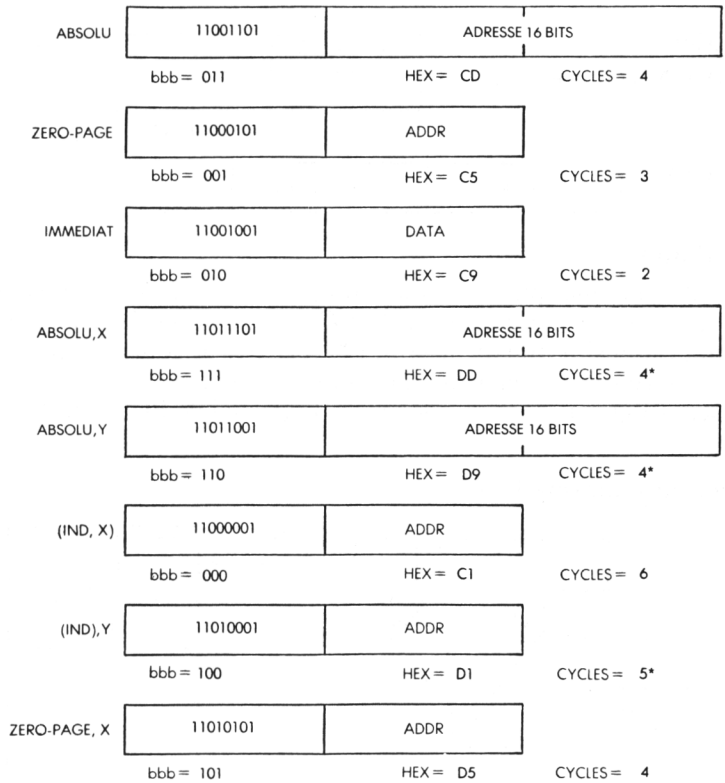
Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0.PAGE	IMMEDIAT	ABS. X	ABS. Y	(IND. X)	(IND). Y	0.PAGE. X	0.PAGE. Y	RELATIF	INDIRECT
HEX			CD	C5	C9	DD	D9	C1	D1	D5			
OCTETS			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

*PLUS 1 CYCLE SI UNE LIMITE DE PAGE EST FRANCHIE

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT	●					●	●

Codes opération :



CPX

Comparer à X

Fonction :

X-DONNEE → NZC :

+ (A >DONNEE)	=	− (A < DONNEE)
−01	011	−00

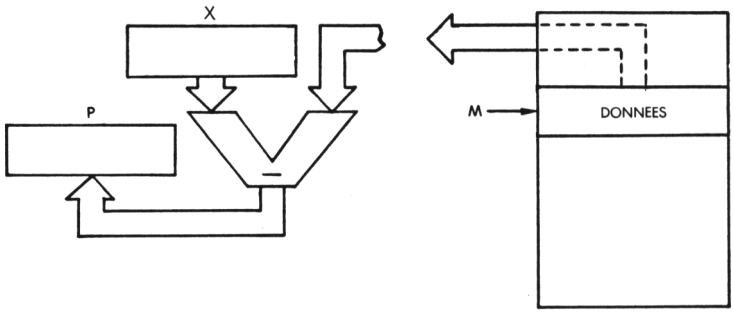
(BINAIRE PUR)

Format :

Description :

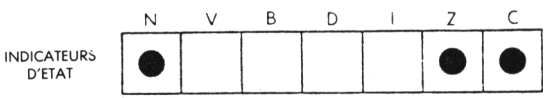
La donnée indiquée est soustraite de X. Le résultat n'est pas stocké, mais les indicateurs N, Z et C sont positionnés selon que le résultat est positif, nul ou négatif. Les registres ne sont pas modifiés. CPX est généralement suivi d'un branchement. BCC détecte A < donnée, BEQ détecte A = donnée et BEQ suivi de BCS détecte A > donnée.

Circuit des informations :

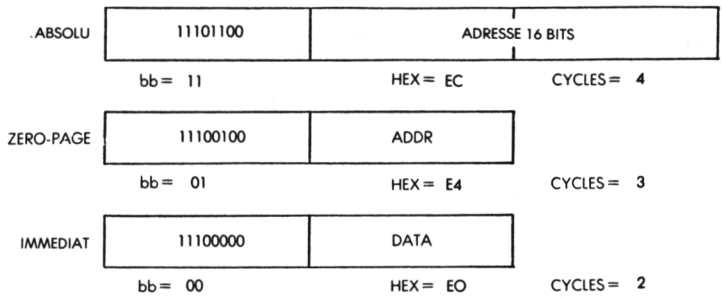


Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0-PAGE	IMMEDIAT	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIF	INDIRECT
HEX			EC	E4	E0								
OCTETS			3	2	2								
CYCLES			4	3	2								
bb			11	01	00								



Codes opération :



CPY

Comparer à Y

Fonction :

(Y) — → NZC :

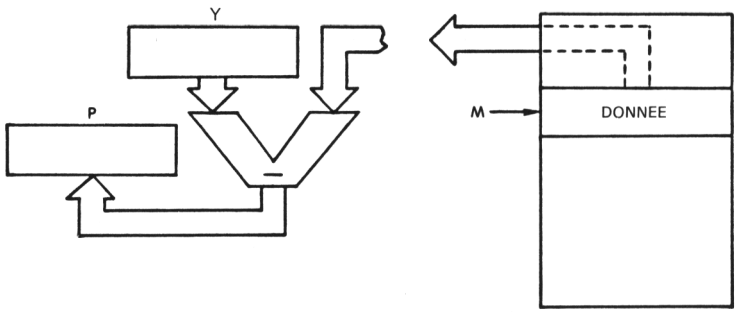
+ (A>DONNEE)	=	— (A<DONNEE)
01	011	00

11ØØbbØØ	ADR/DONNEE	ADR

Description :

La donnée indiquée est soustraite à Y. Le résultat n'est pas stocké mais N, Z et C sont positionnés selon que le résultat est positif, nul ou négatif. Les registres sont inchangés. CPY est généralement suivi d'un branchement : BCC détecte Y < donnée, BEQ détecte Y=donnée et BEQ suivi de BCS détecte Y > donnée.

Circuit des informations :

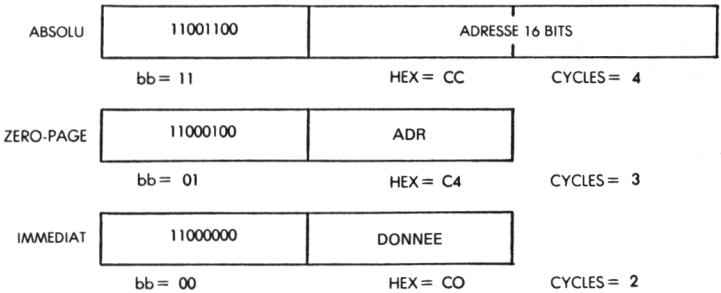


Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0-PAGE	IMMEDIAT	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIF	INDIRECT
HEX			CC	C4	C0								
OCTETS			3	2	2								
CYCLES			4	3	2								
bb			11	01	00								

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT	●					●	●

Codes opération :



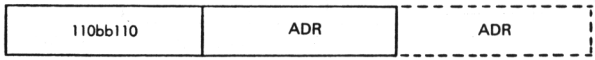
DEC

Décrémenter

Fonction :

M ← (M) — 1

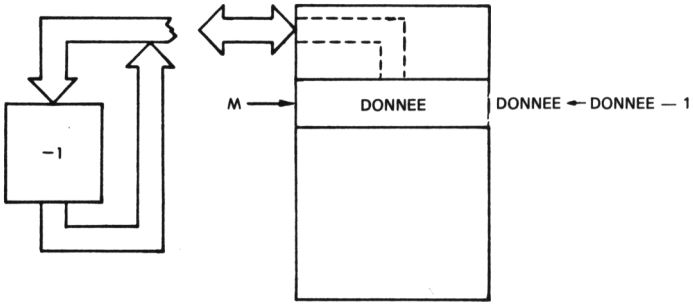
Format :



Description :

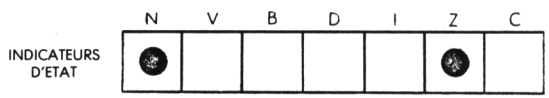
Le contenu de l'adresse spécifiée est diminué de 1. Le résultat est remis à l'adresse spécifiée.

Circuit des informations :

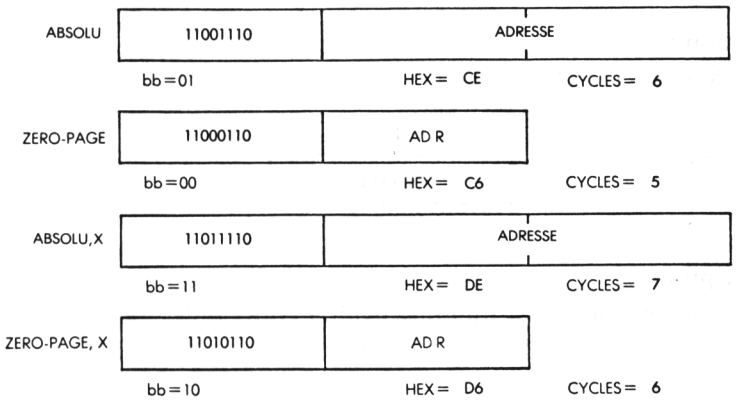


Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0-PAGE	IMMEDIAT	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIF	INDIRECT
HEX			CE	C6		DE				D6			
OCTETS			3	2		3				2			
CYCLES			6	5		7				6			
bb			01	00		11				10			



Codes opération :



DEX

Décrémenter X

Fonction :

$X \leftarrow (X) - 1$

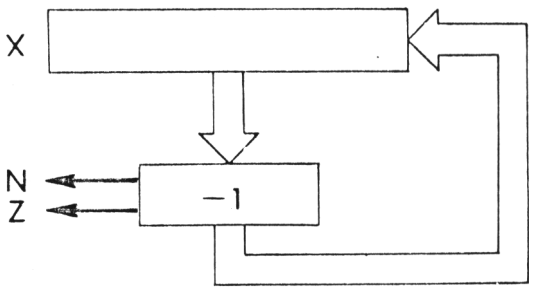
Format :

11001010

Description :

Le contenu de X est diminué de 1. Permet d'utiliser X comme compteur.

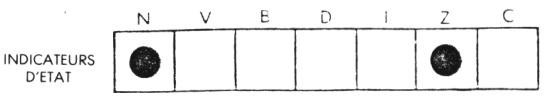
Circuit des informations



Mode d'adressage :

Uniquement implicite.

HEX = CA, octet = 1, cycles = 2



DEY

Décrémenter Y

Fonction :

Y ← (Y) — 1

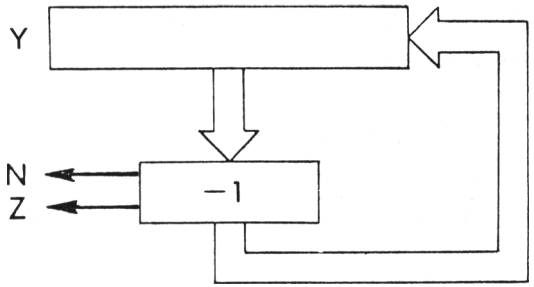
Format :

10001000

Description :

Le contenu de Y est diminué de 1. Permet d'utiliser Y comme compteur.

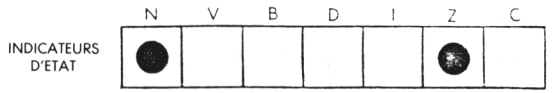
Circuit des informations :



Mode d'adressage :

Uniquement implicite

HEX = 88, octet = 1, cycles = 2

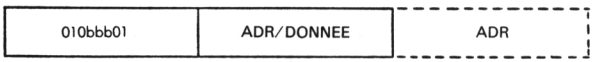


EOR Ou exclusif avec l'accumulateur)

Fonctions :

A ← (A) ∀ DONNEE

Format :



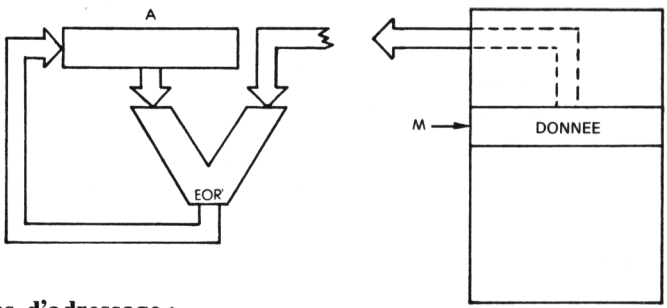
Description :

On effectue le ou exclusif entre la donnée indiquée et l'accumulateur. La table de vérité est :

	0	1
0	0	1
1	1	0

N.B. : EOR avec « 1 » peut servir à complémenter.

Circuit des informations :

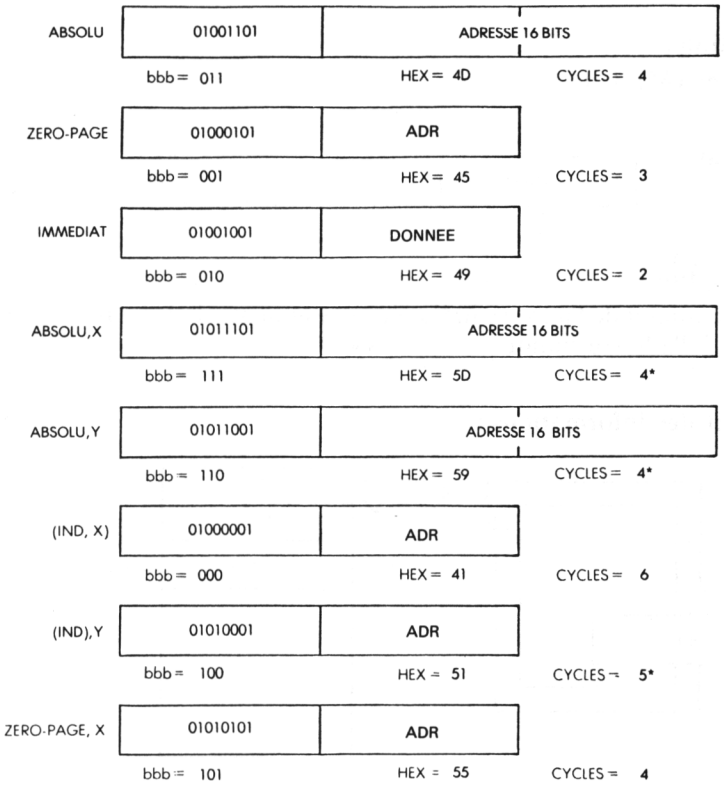


Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0-PAGE	IMMEDIAT	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIF	INDIRECT
HEX			4D	45	49	5D	59	41	51	55			
OCTETS			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT	●					●	

Codes opération :



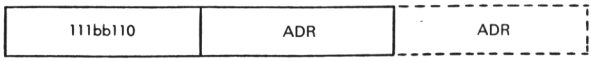
INC

Incrémenter en mémoire

Fonction :

M ← (M) + 1

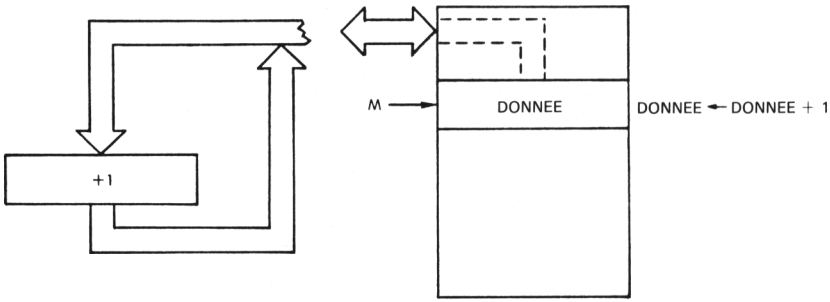
Format :



Description :

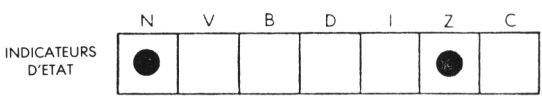
Le contenu de l'adresse indiquée est augmenté de 1. Le résultat est remis à l'adresse indiquée.

Circuit des informations :

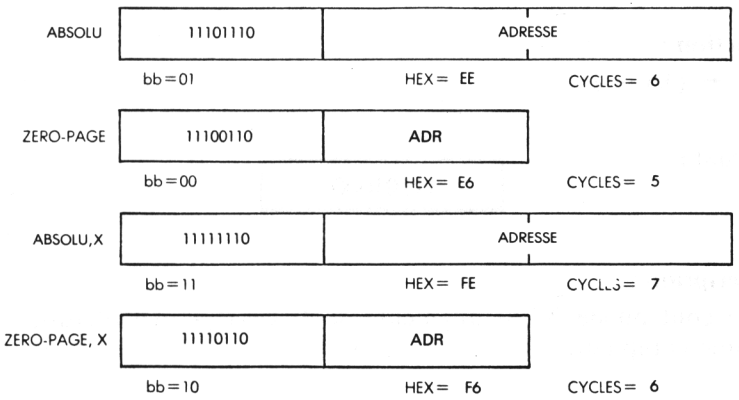


Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0.PAGE	IMMEDIAT	ABS. X	ABS. Y	(IND. X)	(IND). Y	0.PAGE. X	0.PAGE. Y	RELATIF	INDIRECT
HEX			EE	E6		FE			F6				
OCTETS			3	2		3			2				
CYCLES			6	5		7			6				
bb			01	00		11			10				



Codes opération :



INX

Incrémenter X

Fonction :

$X \leftarrow (X) + 1$

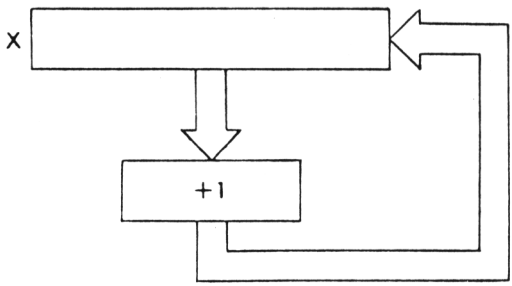
Format :

11101000

Description :

Le contenu de X est augmenté de 1. Ceci permet d'utiliser X comme compteur.

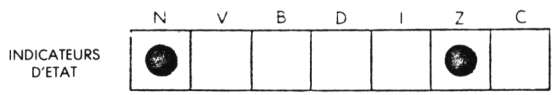
Circuit des informations :



Modes d'adressage :

Uniquement implicite

HEX = E8 octet = 1 cycles = 2



INY

Incrémenter Y

Fonction :

$Y \leftarrow (X) + 1$

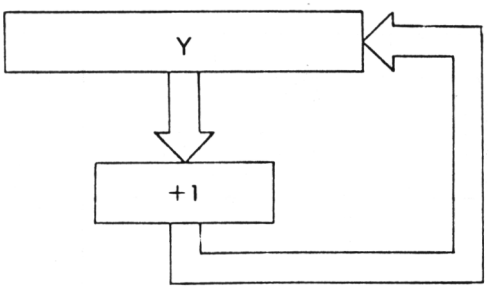
Format :

11001000

Description :

Le contenu de Y est augmenté de 1. Ceci permet d'utiliser Y comme compteur.

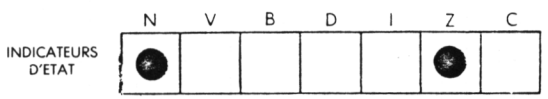
Circuit des informations :



Mode d'adressage :

Uniquement implicite.

HEX = C8, octet = 1, cycles = 2



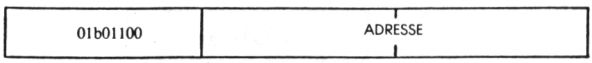
JMP

Saut à une adresse

Fonction :

PC ← ADRESSE

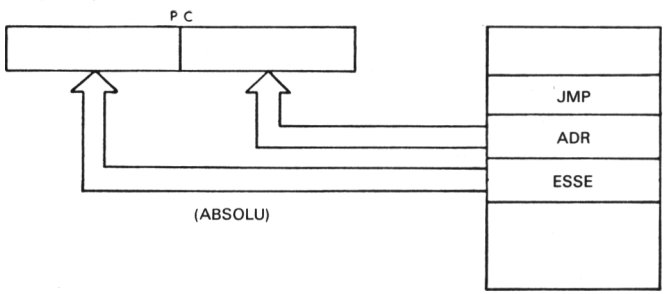
Format :



Description :

On charge une nouvelle adresse dans le compteur ordinal, ce qui entraîne une rupture de séquence du programme. On peut spécifier une adresse absolue ou indirecte.

Circuit des informations :



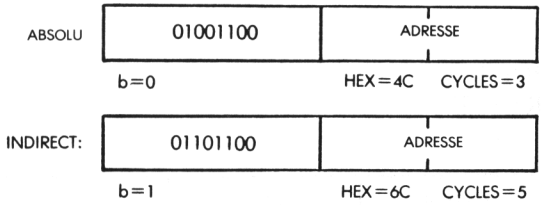
Modes d'adressage :

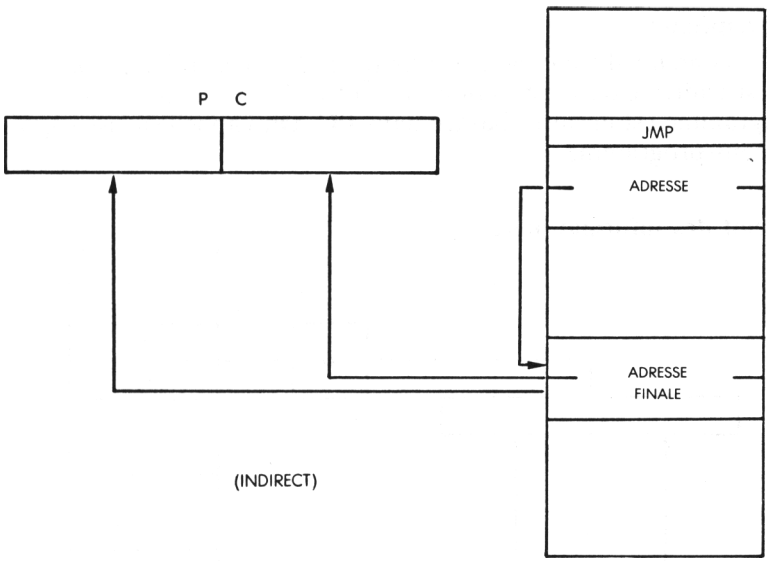
	IMPLICITE	ACCUMU-LATEUR	ABSOLU	0.PAGE	IMMEDIAT	ABS. X	ABS. Y	(IND. X)	(IND.) Y	0.PAGE. X	0.PAGE. Y	RELATIF	INDIRECT
HEX			4C										6C
OCTETS			3										3
CYCLES			3										5
bbb			0										1

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT							

(AUCUN CHANGEMENT)

Codes opération :



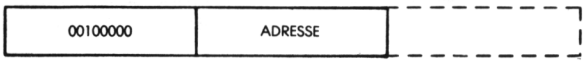


JSR — Saut à un sous-programme

Fonction :

PILE ← (PC) + 2
PC ← ADRESSE

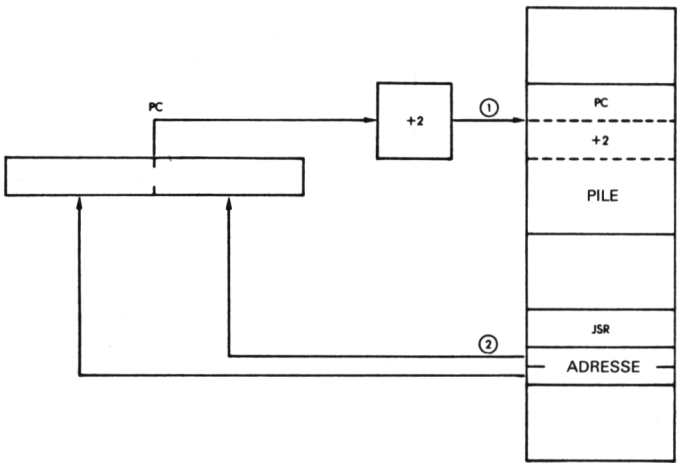
Format :



Description :

Le contenu du compteur ordinal (+ 2) est sauvé dans la pile. (C'est l'adresse de l'instruction qui suit le JSR). L'adresse du sous-programme est ensuite chargée dans PC. On l'appelle aussi « appel de sous-programme ».

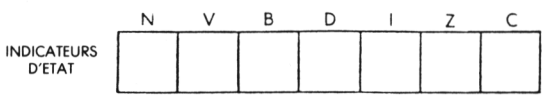
Circuit des informations :



Mode d'adressage :

Uniquement implicite

HEX = 20, octets = 3, cycles = 6



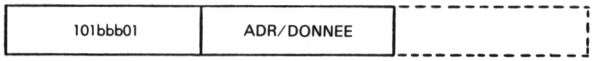
(AUCUN CHANGEMENT)

LDA — Chargement accumulateur

Fonction :

A←DONNEE

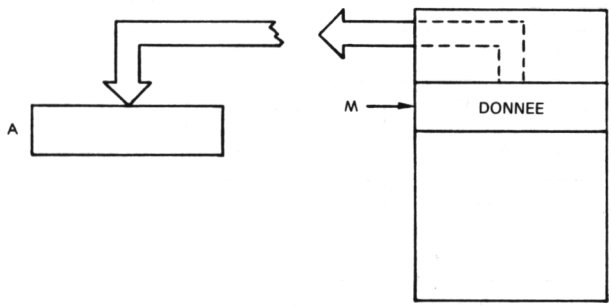
Format :



Description :

On charge une nouvelle donnée dans l'accumulateur.

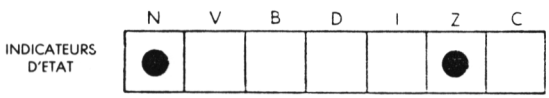
Circuit des informations :



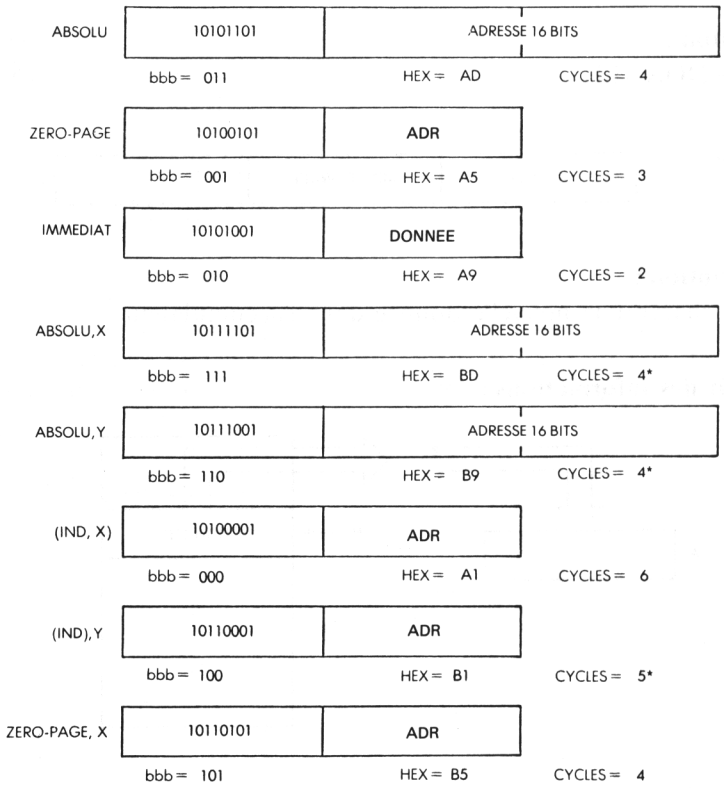
Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0 PAGE	IMMEDIAT	ABS. X	ABS. Y	(IND. X)	(IND). Y	0 PAGE. X	0 PAGE. Y	RELATIF	INDIRECT
HEX			AD	A5	A9	BD	B9	A1	B1	B5			
OCTETS			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

*PLUS 1 CYCLE SI UNE LIMITE DE PAGE EST FRANCHIE



Codes opération :

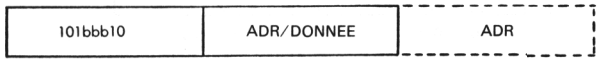


LDX — Chargement du registre X

Fonction :

X ← DONNEE

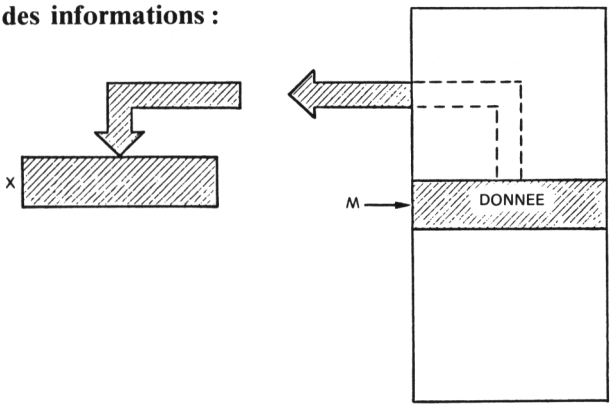
Format :



Description :

On charge le registre X avec la donnée qui se trouve à l'adresse indiquée.

Circuit des informations :



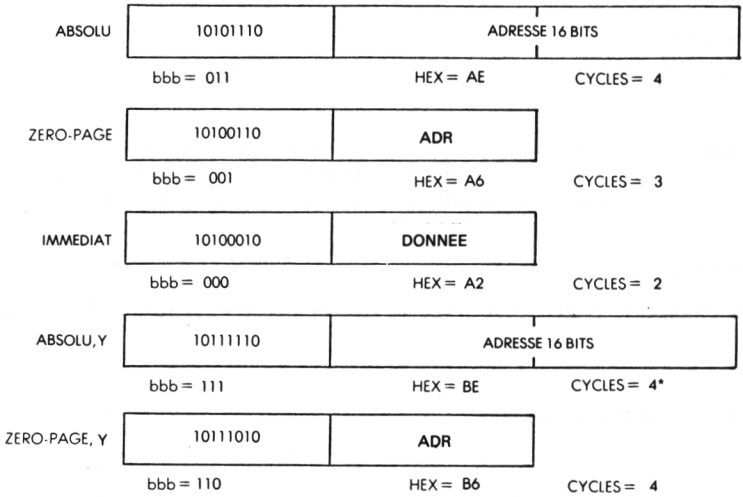
Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0-PAGE	IMMEDIAT	ABS. X	ABS. Y	(IND. X)	(IND). Y	0-PAGE. X	0-PAGE. Y	RELATIF	INDIRECT
HEX			AE	A6	A2		BE				B6		
OCTETS			3	2	2		3				2		
CYCLES			4	3	2		4*				4		
bbb			011	001	000		111				110		

*PLUS 1 CYCLE SI UNE LIMITE DE PAGE EST FRANCHIE

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT	●					●	

Codes opération :

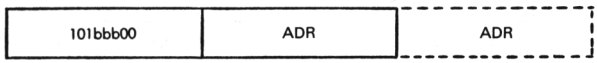


LDY — Chargement du registre Y

Fonction :

Y ← DONNEE

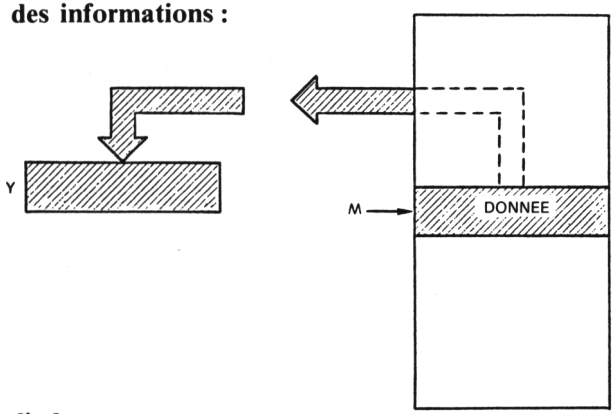
Format :



Description :

On charge le registre Y avec la donnée trouvée à l'adresse indiquée.

Circuit des informations :



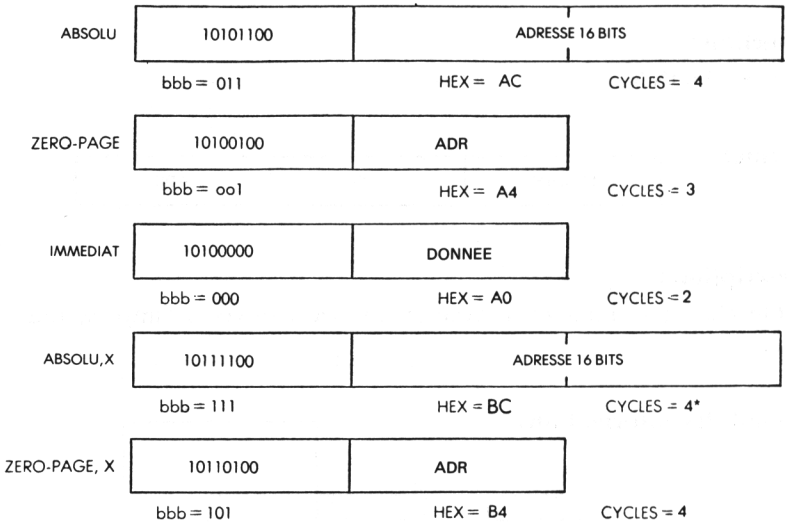
Modes d'adressage :

	IMPLICITE	ACCUMU-LATEUR	ABSOLU	0-PAGE	IMMEDIAT	ABS. X	ABS. Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIF	INDIRECT
HEX			AC	A4	A0	BC				B4			
OCTETS			3	2	2	3				4			
CYCLES			4	3	2	4*				4			
bbb			011	001	000	111				101			

* PLUS 1 CYCLE SI UNE LIMITE DE PAGE EST FRANCHIE

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT	●					●	

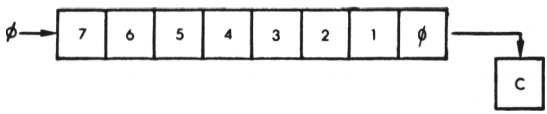
Codes opération :



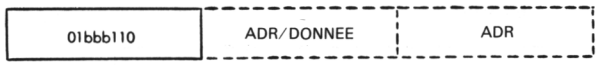
LSR

Décalage logique à droite

Fonction :



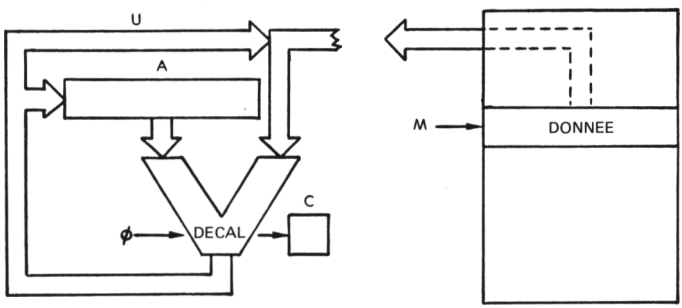
Format :



Description :

On décale d'un cran à droite l'emplacement indiqué (mémoire ou accumulateur). Un 0 est forcé dans le bit 7. Le bit 0 est transféré dans la retenue. La donnée décalée est renvoyée à la source (accumulateur ou mémoire).

Circuit des informations :

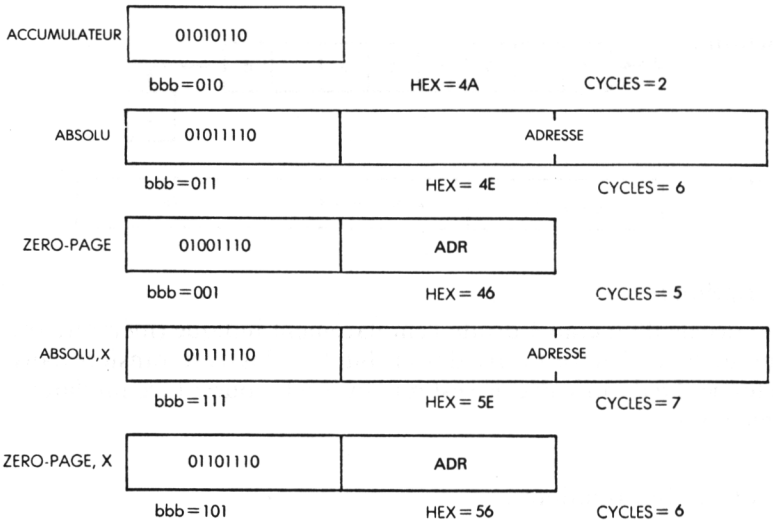


Mode d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0-PAGE	IMMEDIAT	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIF	INDIRECT
HEX		4A	4E	46		5E				56			
OCTETS		1	3	2		3				2			
CYCLES		2	6	5		7				6			
bbb		010	011	001		111				101			

INDICATEURS D'ETAT	N	V	B	D	I	Z	C
	∅					●	●

Codes opération :



NOP

Instruction muette

Fonction :

Aucune

Format :

11101010

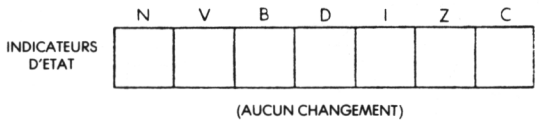
Description :

On ne fait rien mais cela dure deux cycles. Peut servir pour rallonger des boucles de délai, ou pour remplir des trous créés par des corrections de programme.

Mode d'adressage :

Uniquement implicite

HEX = EA, octet = 1, cycles = 2



ORA — Ou inclusif avec l'accumulateur

Fonction :

A ← (A) V DONNEE

Format :

000bbb01	ADR/DONNEE

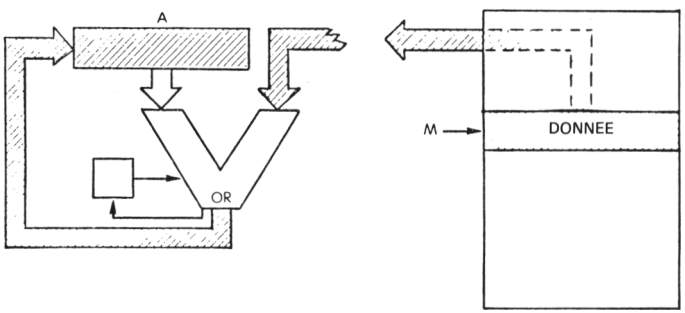
Description :

On effectue le OU inclusif de A et de la donnée indiquée. Le résultat est rangé en A. Cela peut servir à forcer à 1 des bits choisis.

Table de vérité.

	0	1
0	0	1
1	1	1

Circuit des informations :



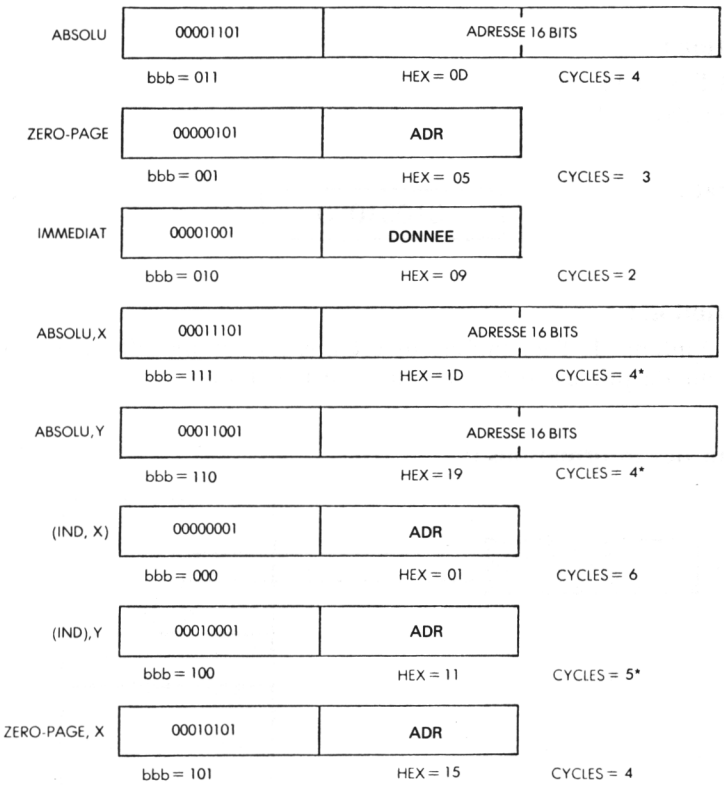
Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0-PAGE	IMMEDIAT	ABS. X	ABS. Y	(IND. X)	(IND.), Y	0-PAGE, X	0-PAGE, Y	RELATIF	INDIRECT
HEX			0D	05	09	1D	19	01	11	15			
OCTETS			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

*PLUS 1 CYCLE SI UNE LIMITE DE PAGE EST FRANCHIE

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT	●					●	

Codes opération :



PHA

Empiler A

Fonction :

PILE ← (A)
S ← (S) — 1

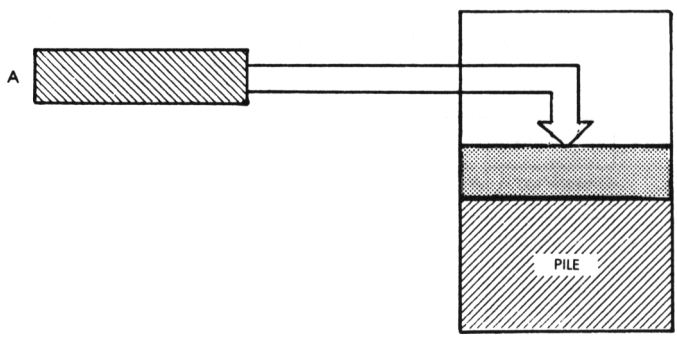
Format :

01001000

Description :

Le contenu de l'accumulateur est mis au sommet de la pile. Le pointeur de pile est remis à jour. A est inchangé.

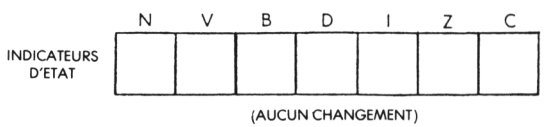
Circuit des informations :



Mode d'adressage :

Uniquement implicite

HEX = 48, octet = 1, cycles = 3



PHP

Empiler P

Fonction :

PILE ← (P)
S ← (S) — 1

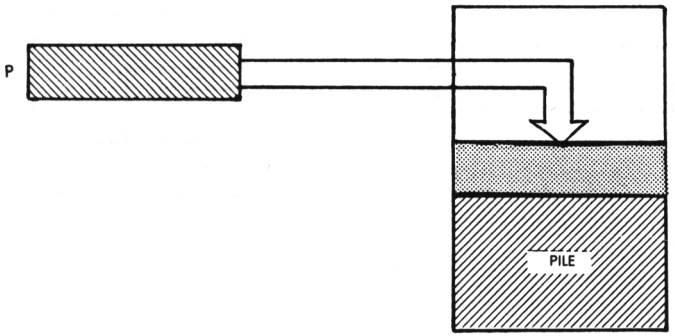
Format :

00001000

Description :

Le contenu du registre d'état P est mis au sommet de la pile. Le pointeur de pile est remis à jour. P est inchangé.

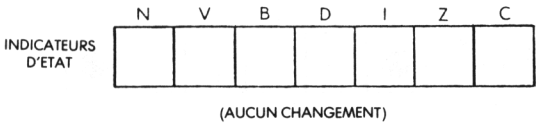
Circuit des informations :



Mode d'adressage :

Uniquement implicite.

HEX = 08, octet = 1, cycles = 3



PLA

Dépiler l'accumulateur

Fonction :

A ← (PILE)
S ← (S) + 1

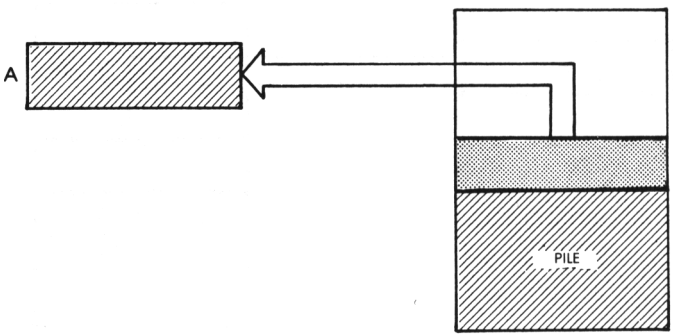
Format :

01101000

Description :

Le sommet de la pile est transféré vers l'accumulateur. Le pointeur de pile est incrémenté.

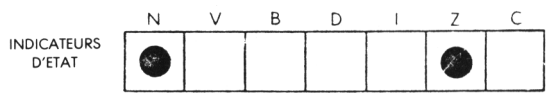
Circuit des informations :



Mode d'adressage :

Uniquement implicite

HEX = 68, octet = 1, cycles = 4



PLP

Dépiler le registre d'état P

Fonction :

P ← (PILE)
P ← (S) + 1

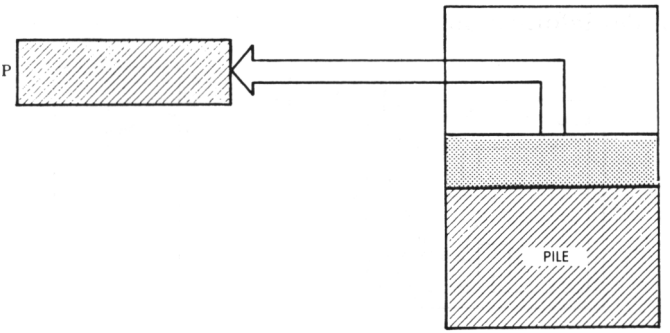
Format :

00101000

Description :

Le sommet de la pile est transféré vers le registre d'état P. Le pointeur de pile est incrémenté.

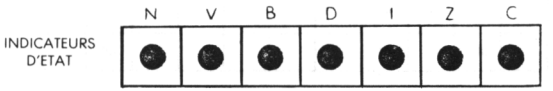
Circuit des informations :



Mode d'adressage :

Uniquement implicite

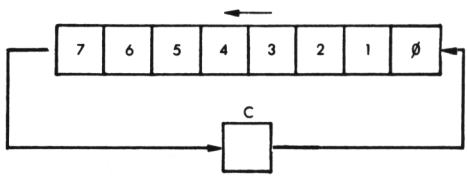
HEX = 28, octet = 1, cycles = 4



ROL

Rotation à gauche

Fonction :

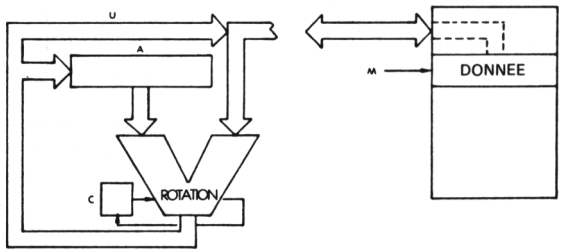


Format :

Description :

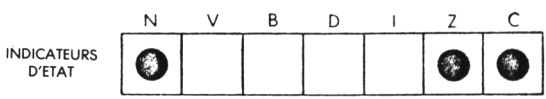
Le contenu de l'adresse indiquée (accumulateur de mémoire) est tourné d'un cran à gauche. La retenue entre dans le bit 0. Le bit 7 devient la nouvelle valeur de la retenue. C'est donc une rotation sur 9 bits.

Circuit des informations :

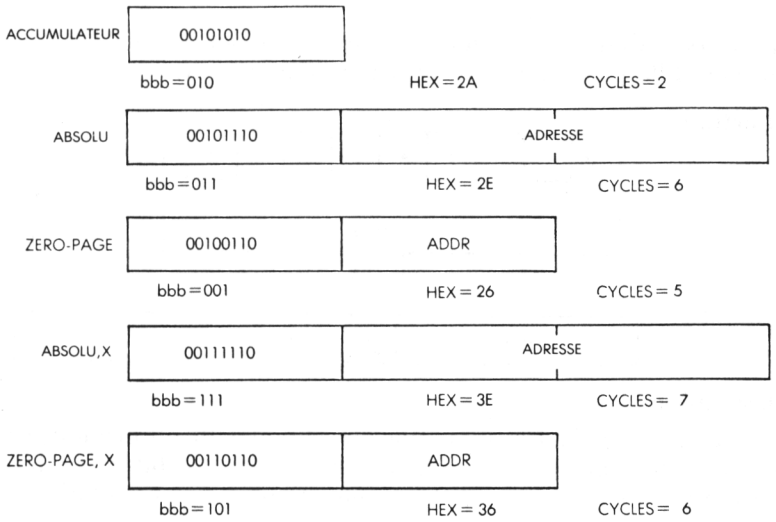


Mode d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0-PAGE	IMMEDIAT	ABS. X	ABS. Y	(IND. X)	(IND). Y	0-PAGE. X	0-PAGE. Y	RELATIF	INDIRECT
HEX		2A	2E	26		3E				36			
OCTETS		1	3	2		3				2			
CYCLES		2	6	5		7				6			
bbb		010	011	001		111				101			



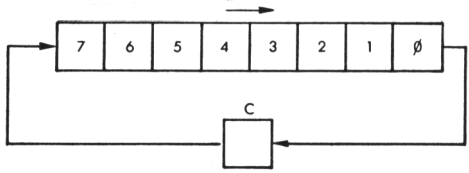
Codes opération :



ROR — Rotation à droite

Attention : Cette instruction peut manquer sur les 6502 les plus anciens. Elle peut aussi être présente, mais non listée.

Fonction :

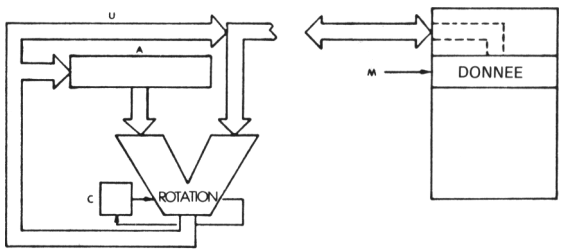


Format :

Description :

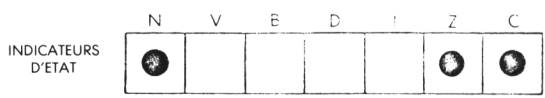
Le contenu de l'adresse indiquée (accumulateur ou mémoire) est tourné d'un cran à droite. La retenue va dans le bit 7. Le bit 0 devient la nouvelle valeur de la retenue. C'est donc une rotation sur 9 bits.

Circuit des informations :

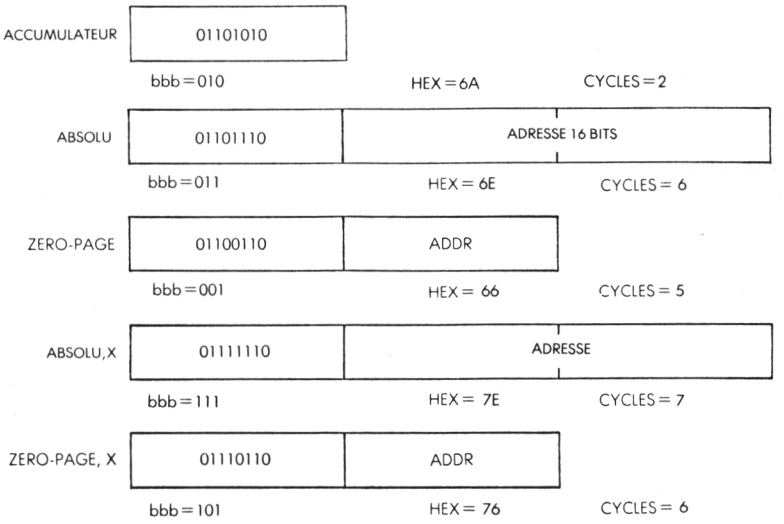


Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0 PAGE	IMMEDIAT	ABS, X	ABS, Y	(IND, X)	(IND), Y	0 PAGE, X	0 PAGE, Y	RELATIF	INDIRECT
HEX		6A	6E	66		7E				76			
OCTETS		1	3	2		3				2			
CYCLES		2	6	5		7				6			
bbb		010	011	001		111				101			



Codes opération :



RTI

Retour d'interruption

Fonction :

P ← (PILE)
S ← (S) + 1
PCL ← (PILE)
S ← (S) + 1
PCH ← (PILE)
S ← (S) + 1

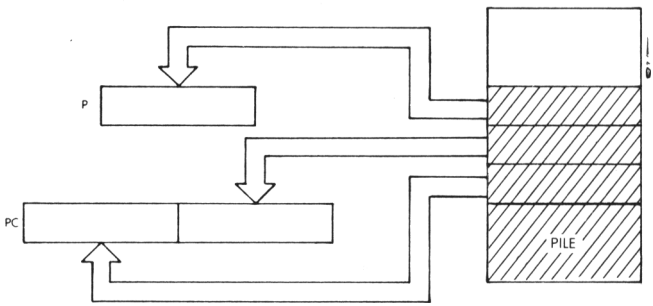
Format :

01000000

Description :

Restaure le registre d'état P et le compteur ordinal (PC) qui avaient été sauvés dans la pile. Remet à jour le pointeur de pile.

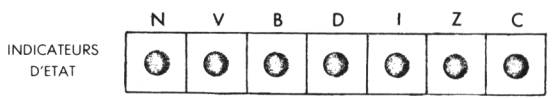
Circuit des informations :



Mode d'adressage :

Uniquement implicite

HEX = 40, octet = 1, cycles = 6



RTS — Retour de sous-programme

Fonction :

PC ← (PILE) (2 octets)
PC ← (PC) + 1
S ← (S) + 2

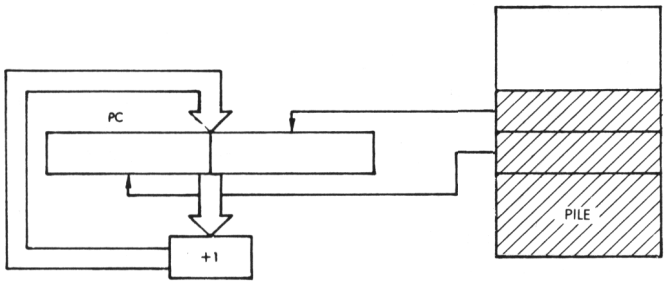
Format :

01100000

Description :

Restaure le compteur ordinal à partir de la pile et l'augmente de 1. Remet à jour le pointeur de pile.

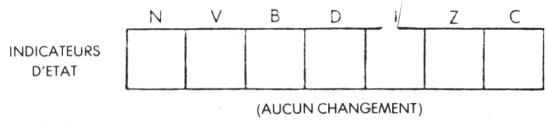
Circuit des informations :



Modes d'adressage :

Uniquement implicite

HEX = 60, octet = 1, cycles = 6



SBC

Soustraction avec retenue

Fonction :

A ← (A) — DONNEE — C (C est l'emprunt)

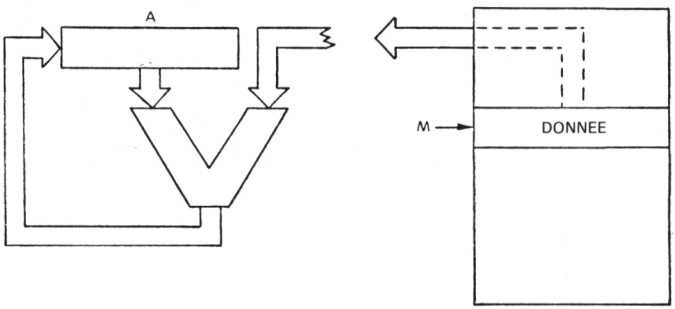
Format :

Description :

On soustrait de l'accumulateur la donnée à l'adresse indiquée, avec l'emprunt. Le résultat est mis dans A. N.B. : pour effectuer une soustraction sans emprunt, on utilise SEC.

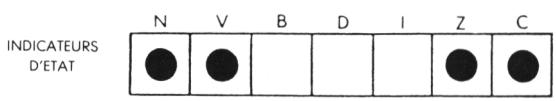
SBC peut être utilisé en mode décimal ou binaire, selon la valeur du bit D du registre d'état.

Circuit des informations :

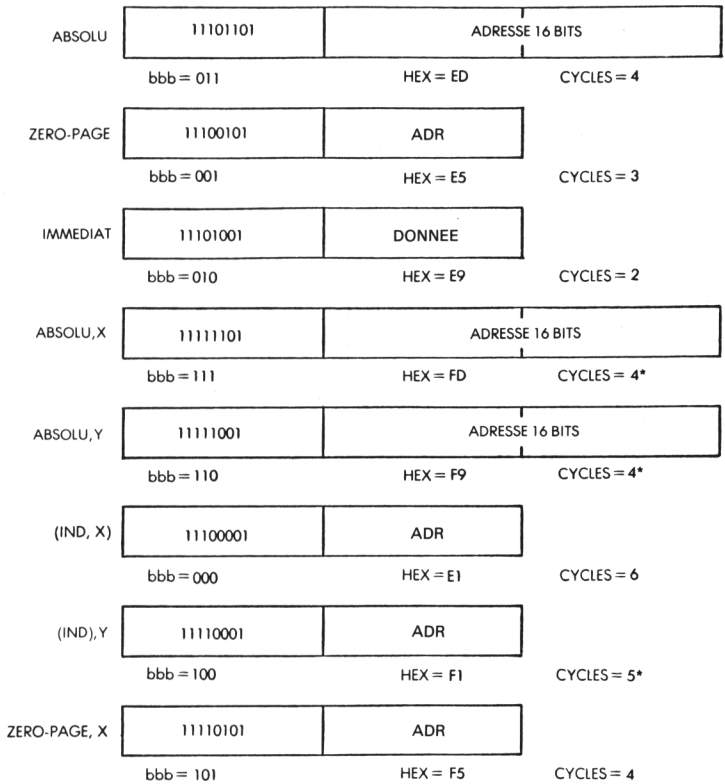


Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0.PAGE	IMMEDIAT	ABS. X	ABS. Y	(IND. X)	(IND.), Y	0.PAGE. X	0.PAGE. Y	RELATIF	INDIRECT
HEX			ED	E5	E9	FD	F9	E1	F1	F5			
OCTETS			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			



Codes opération :



SEC

Mise à 1 de la retenue

Fonction :

C ← 1

Format :

00111000

Description :

Le bit de retenue est forcé à 1. C'est utilisé avant un SBC pour effectuer une soustraction sans retenue.

Mode d'adressage :

Uniquement implicite

HEX = 38, octet = 1, cycles = 2

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT							1

SED

Mise en mode décimal

Fonction :

D ← 1

Format :

11111000

Description :

On met à 1 le bit D du registre d'état. Lorsque celui-ci est à 0, on est en mode binaire. Lorsqu'il est à 1, on est en mode décimal pour les instructions ADC et SBC.

Mode d'adressage :

Uniquement implicite

HEX = F8, octet = 1, cycles = 2

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT				1			

SEI — Mise à un de l'inhibition interruptions

Fonction :

I ← 1

Format : 01111000

Description :

On met à 1 le masque des interruptions. C'est utilisé lors d'une interruption ou d'une réinitialisation du système.

Mode d'adressage :

Uniquement implicite

HEX = 78, octet = 1, cycles = 2

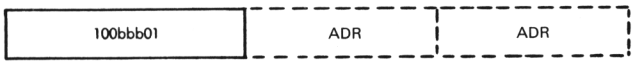
	N	V	B	D	I	Z	C
INDICATEURS D'ETAT					1		

STA Rangement de l'accumulateur en mémoire

Fonction :

M ← (A)

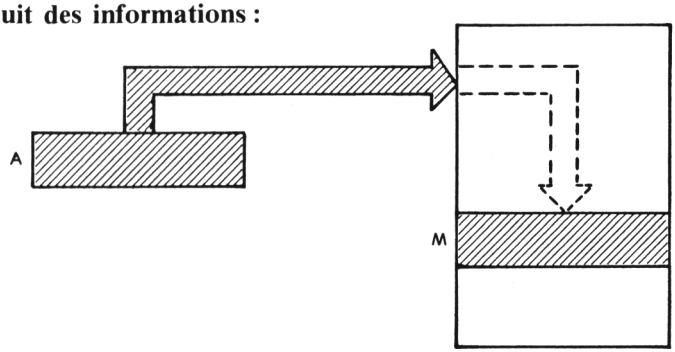
Format :



Description :

Le contenu de A est copié dans l'emplacement mémoire indiqué. Le contenu de A est inchangé.

Circuit des informations :



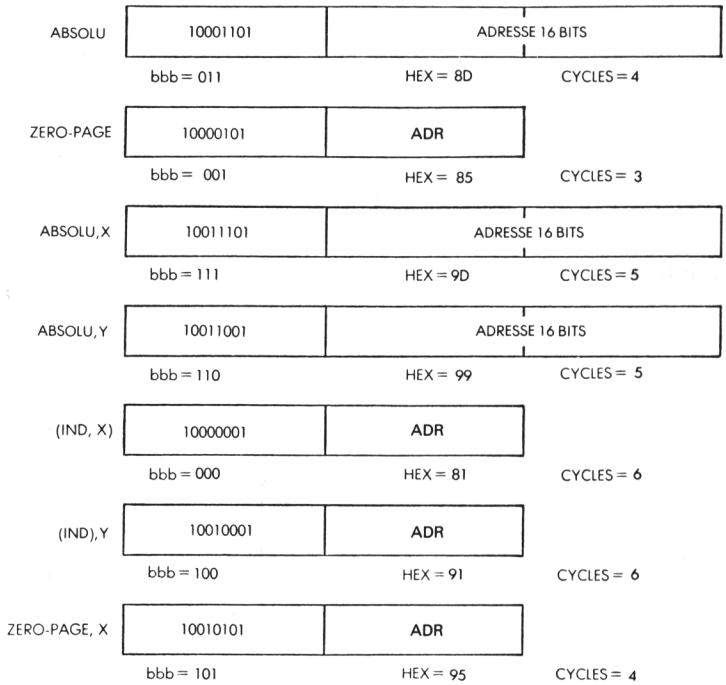
Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0-PAGE	IMMEDIAT	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIF	INDIRECT
HEX			8D	85		9D	99	81	91	95			
OCTETS			3	2		3	3	2	2	2			
CYCLES			4	3		5	5	6	6	4			
bbb			011	001		111	110	000	100	101			

	N	V	B	D	I	Z	C
INDICATEURS D'ETAT							

(AUCUN CHANGEMENT)

Codes opération :



STX — Ranger X en mémoire

Fonction :

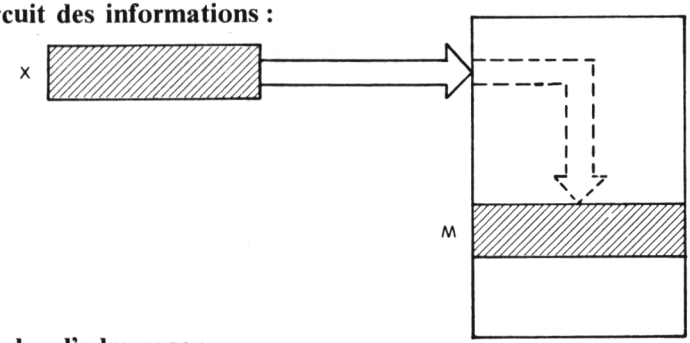
M ← (X)

Format :

Description :

Le contenu du registre index X est copié dans l'emplacement mémoire indiqué. Le contenu de X est inchangé.

Circuit des informations :



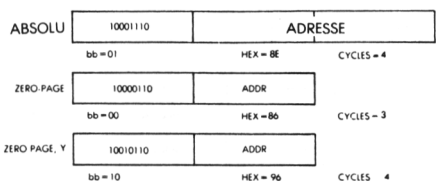
Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0.PAGE	IMMEDIAT	ABS. X	ABS. Y	(IND. X)	(IND). Y	0.PAGE. X	0.PAGE. Y	RELATIF	INDIRECT
HEX			8E	86							96		
OCTETS			3	2							2		
CYCLES			4	3							4		
bb			01	00							10		

INDICATEURS D'ETAT	N	V	B	D	I	Z	C

(AUCUN CHANGEMENT)

Codes opération :



STY — Ranger Y en mémoire

Fonction :

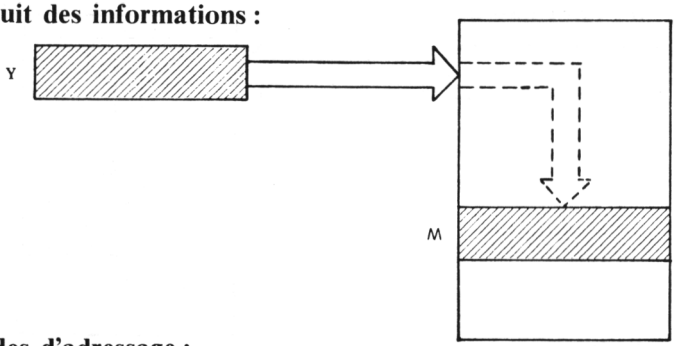
M ← (Y)

Format :

Description :

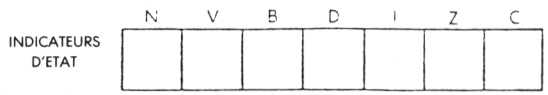
Le contenu du registre index Y est copié dans l'emplacement mémoire indiqué. Le contenu de Y est inchangé.

Circuit des informations :



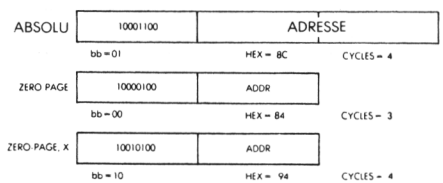
Modes d'adressage :

	IMPLICITE	ACCUMULATEUR	ABSOLU	0-PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIF	INDIRECT
HEX			8C	84						94			
OCTETS			3	2						4			
CYCLES			4	3						4			
bb			01	00						10			



AUCUN CHANGEMENT

Codes opération :



TAX

Transfert de A dans X

Fonction :

X ← (A)

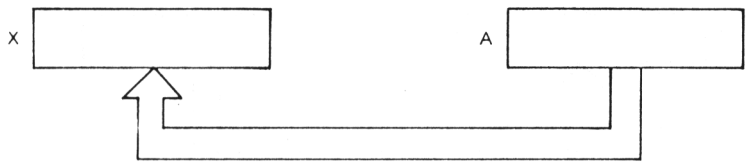
Format :

10101010

Description :

On copie le contenu de l'accumulateur dans le registre index X. Le contenu de A est inchangé.

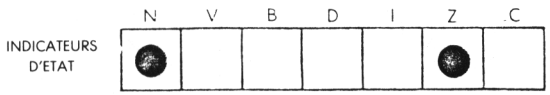
Circuit des informations :



Mode d'adressage :

Uniquement implicite

HEX = AA, octet = 1, cycles = 2



TAY

Transfert de A dans Y

Fonction :

Y ← (A)

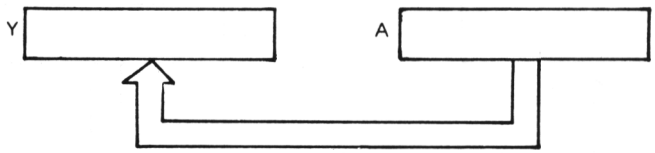
Format :

10101000

Description :

On copie le contenu de l'accumulateur dans le registre index Y. Le contenu de A est inchangé.

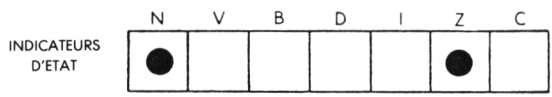
Circuit des informations :



Mode d'adressage :

Uniquement implicite

HEX = A8, octet = 1, cycles = 2



TSX — Transfert de S dans X

Fonction :

X ← (S)

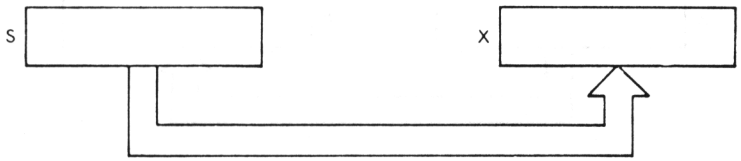
Format :

10111010

Description :

Le contenu du pointeur de pile S est copié dans le registre index X. Le contenu de S est inchangé.

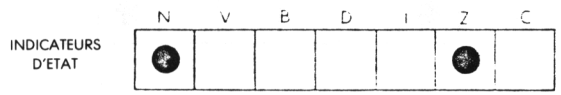
Circuit des informations :



Mode d'adressage :

Uniquement implicite

HEX = BA, octet = 1, cycles = 2



TXA Transfert de X dans l'accumulateur

Fonction :

A ← (X)

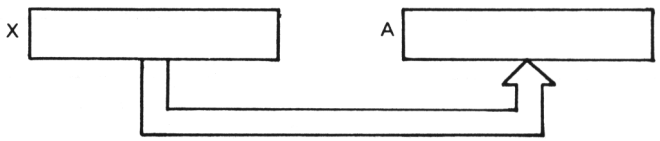
Format :

10001010

Description :

Le contenu du registre index X est copié dans l'accumulateur. Le contenu de X est inchangé.

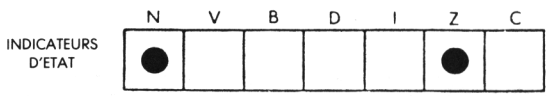
Circuit des informations :



Mode d'adressage :

Uniquement implicite

HEX = 8A, octet = 1, cycles = 2



TXS — Transfert de X dans S

Fonction :

S ← (X)

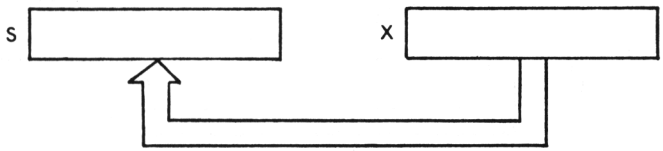
Format :

10011010

Description :

Le contenu du registre index X est copié dans le pointeur de pile S. Le contenu de X est inchangé.

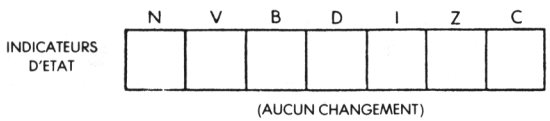
Circuit des informations :



Mode d'adressage :

Uniquement implicite

HEX = 9A, octet = 1, cycles = 2



TYA — Transfert de Y dans A

Fonction :

A ← (Y)

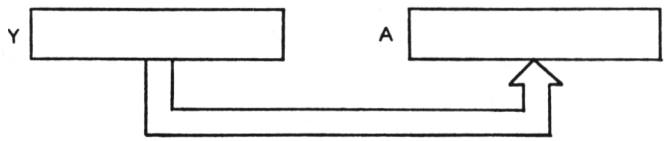
Format :

10011000

Description :

Le contenu du registre index Y est copié dans l'accumulateur. Le contenu de Y est inchangé.

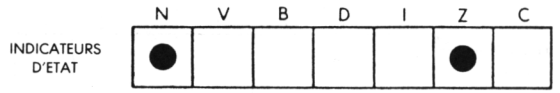
Circuit des informations :



Mode d'adressage :

Uniquement implicite

HEX = 98, octet = 1, cycles = 2



5

TECHNIQUES D'ADRESSAGE

INTRODUCTION

Ce chapitre va présenter la théorie générale de l'adressage, ainsi que les diverses techniques qui ont été développées pour faciliter l'accès aux données. Dans la deuxième section, les modes d'adressage spécifiques dont dispose le 6502 seront passés en revue avec leurs avantages et leurs limitations là où elles existent. Enfin, pour familiariser le lecteur avec les divers compromis possibles, une section d'applications montrera dans le cadre des programmes d'applications spécifiques, les choix possibles entre les différentes techniques d'adressage.

Comme le 6502 n'a pas d'autre registre à 16 bits que le compteur ordinal pour spécifier une adresse, il est nécessaire que tout utilisateur du 6502 comprenne bien les différents modes d'adressage et, en particulier, l'utilisation des registres index. Les modes élaborés comme la combinaison de l'adressage indirect et de l'adressage indexé peuvent être laissés de côté au début. Cependant, tous les modes d'adressage sont utiles pour développer des programmes efficaces pour ce microprocesseur. Etudions maintenant les diverses alternatives offertes.

MODES D'ADRESSAGE

L'adressage désigne la spécification dans une instruction de l'emplacement de l'opérande sur lequel l'instruction va opérer. Les principales méthodes d'adressage vont être maintenant examinées.

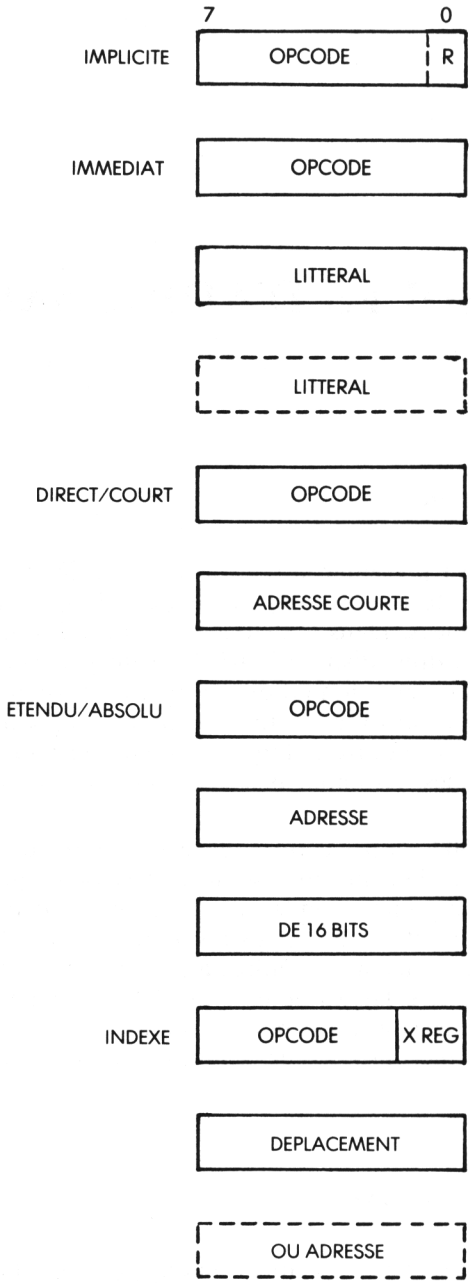


Fig. 5-1 : Adressage

Adressage implicite

Les instructions qui n'opèrent que sur des registres internes utilisent normalement l'*adressage implicite*. Ceci est illustré sur la Figure 5-1. Le nom dérive du fait que l'instruction ne contient pas explicitement l'adresse de l'opérande sur lequel elle agit. A la place, son code opération spécifie (implicitement) un ou plusieurs registres (généralement l'accumulateur ou un autre). Comme les registres internes sont habituellement peu nombreux (disons au maximum 8), cela nécessite un petit nombre de bits. Par exemple, trois bits de l'instruction désigneront un registre interne parmi 8. De telles instructions peuvent être par conséquent codées seulement en 8 bits. Ceci est un avantage important, car une instruction sur un octet s'exécute généralement plus vite qu'une instruction sur deux ou trois octets. Une instruction implicite du 6502 est, par exemple : TAX qui signifie « transférer le contenu de A dans X ».

Adressage immédiat

L'adressage immédiat est illustré sur la Figure 5-1. Le code opération sur 8 bits est suivi par un littéral (constante) sur 8 ou 16 bits. Ce type d'instruction est nécessaire, par exemple, pour charger une valeur de 8 bits dans un registre 8 bits.

Si le microprocesseur possède des registres 16 bits, il peut être nécessaire de charger des constantes 16 bits. Cela dépend de l'architecture interne du processeur. Un exemple d'instruction immédiate du 6502 est ADC # 0.

Le second mot de cette instruction contient le littéral « 0 » qui est ajouté à l'accumulateur.

Adressage absolu

L'adressage absolu désigne la manière normale de retrouver des données en mémoire, selon laquelle un code opération est suivi d'une adresse à 16 bits. L'adressage absolu nécessite donc 3 octets pour l'instruction. Un exemple d'adressage absolu est : STA $1234.

Cette instruction spécifie de ranger le contenu de l'accumulateur à l'adresse mémoire « 1234 » en hexadécimal.

L'inconvénient de l'adressage absolu est qu'il demande des instructions sur 3 octets. Pour permettre une meilleure efficacité, un autre mode d'adressage est à la disposition de l'utilisateur, où l'adresse n'occupe qu'un octet : l'adressage direct.

Adressage direct

Dans ce mode d'adressage, le code opération est suivi par une adresse sur 8 bits. Ceci est illustré sur la Figure 5-1. L'avantage de cette approche est de ne nécessiter que 2 octets au lieu de 3 comme l'adressage absolu. L'inconvénient est de limiter les adresses à la gamme 0-255, c'est-à-dire la page 0. Ce mode est aussi appelé adressage court, ou adressage en page 0. Quand l'adressage court est disponible, on qualifie souvent l'adressage absolu d'*adressage étendu,* par contraste.

Adressage relatif

Les instructions de saut normal requièrent 8 bits pour le code opération, plus les 16 bits de l'adresse à laquelle on veut sauter.

Exactement comme dans le cas précédent, ceci a l'inconvénient de demander 3 mots, donc au moins 3 cycles mémoire. Pour fournir des branchements plus efficaces, l'adressage relatif fait appel à un format sur deux mots seulement. Le premier mot spécifie le branchement, généralement accompagné du test correspondant. Le second mot est un déplacement. Comme le déplacement doit être positif ou négatif, une instruction de branchement relatif permet un branchement en avant de 128 octets (7 bits) ou un branchement en arrière de 128 octets (plus ou moins 1 selon les conventions). Comme la plupart des boucles tendent à être courtes, les branchements relatifs peuvent être utilisés dans la plupart des cas, et il en résulte des performances considérablement améliorées pour des routines courtes de cette sorte. Par exemple, nous avons déjà utilisé l'instruction BCC qui spécifie un « branchement si retenue nulle » à un emplacement situé à moins de 127 octets de distance de l'instruction de branchement.

Adressage indexé

L'adressage indexé est une technique spécialement utile pour accéder successivement aux éléments d'un bloc ou d'une table. Cela sera illustré par des exemples plus loin dans ce chapitre. Le principe de l'adressage indexé est que l'instruction spécifie à la fois un registre index et une adresse. Dans le mode le plus général, le contenu du registre est ajouté à l'adresse pour fournir l'adresse finale. De cette manière, l'adresse pourrait être le début d'une table en mémoire. Le registre index serait alors utilisé pour accéder successivement à tous les éléments de la table de façon efficace. En pratique, il y a souvent des restrictions qui peuvent limiter la taille du registre index ou celle de la zone adresse ou déplacement.

Pré-indexation et post-indexation

Deux modes d'indexation peuvent être distingués. La pré-indexation est le mode d'indexation habituel où l'adresse finale est la somme d'une adresse ou d'un déplacement et du contenu du registre index.

Le post-indexation considère le contenu du champ déplacement comme *l'adresse* du déplacement réel et non comme le déplacement lui-même. Ceci est illustré sur la Figure 5-2. En adressage post-indexé, l'adresse finale est la somme du contenu du registre index plus le contenu du mot mémoire *désigné par la zone déplacement.* Ceci utilise en fait une combinaison de la pré-indexation et de l'adressage indirect. Mais nous n'avons pas encore défini l'adressage indirect. Faisons le.

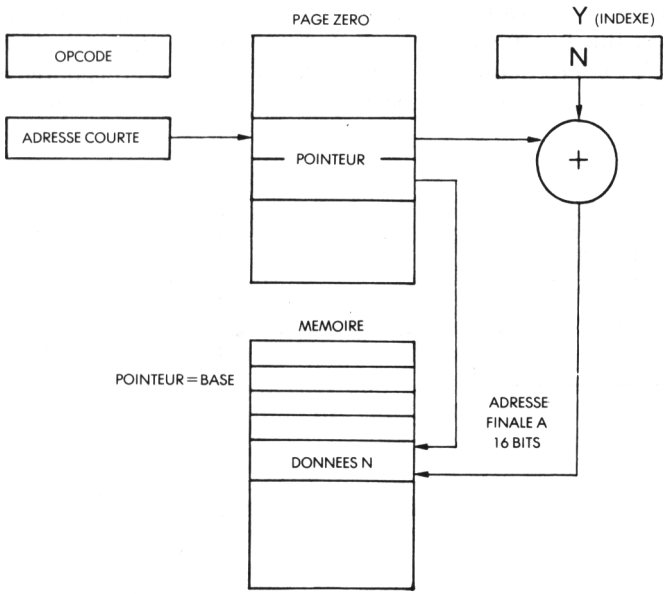


Fig. 5-2 : Adressage indirect post-indexé

Adressage indirect

Nous avons déjà vu le cas où deux sous-programmes souhaitent échanger une grande quantité de données stockées en mémoire. Plus généralement, plusieurs programmes, ou plusieurs sous-programmes, peuvent avoir besoin d'accéder à un bloc d'informations

commun. Pour conserver la généralité du programme, il est désirable de ne pas garder un tel bloc à une adresse-mémoire fixe. En particulier la taille de ce bloc peut grandir ou diminuer de manière dynamique, et il peut avoir à résider dans différentes zones de la mémoire, en fonction de sa taille. Il serait donc impraticable d'avoir à accéder à ce bloc avec des adresses absolues. La solution de ce problème est de déposer l'adresse de début du bloc à un emplacement-mémoire fixe. Ceci est analogue à la situation où plusieurs personnes ont besoin d'entrer dans une maison alors qu'il n'y a qu'une clé. Par convention, la clé sera déposée sous le paillasson. Chacun sait alors où regarder (sous le paillasson) pour trouver la clé de la maison (ou, pour une analogie plus exacte, pour trouver l'adresse de la réunion prévue). L'adressage indirect met donc en jeu un code opération de 8 bits suivi d'une adresse 16 bits. Cette adresse est simplement utilisée pour retrouver un mot-mémoire. Normalement, ce sera un mot de 16 bits (dans notre cas 2 octets), dans la mémoire. Ceci est illustré sur la Figure 5-3. Les deux octets à l'adresse spécifiée « A 1 », contiennent « A 2 ». « A 2 » est alors considérée comme l'adresse effective des données que l'on souhaite manipuler.

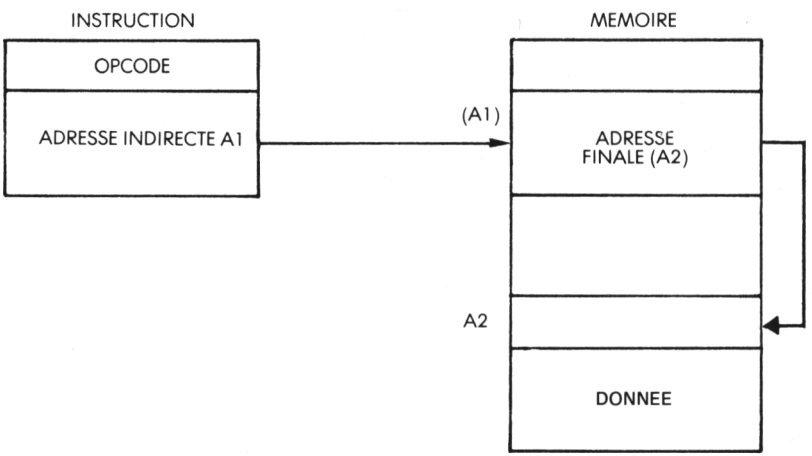


Fig. 5-3 : Adressage indirect

L'adressage indirect est particulièrement utile chaque fois qu'on utilise des pointeurs. Différentes zones du programme peuvent alors faire appel à ces pointeurs pour accéder à un mot ou un bloc de données d'une façon commode et élégante.

Combinaisons de modes

Les modes ci-dessus peuvent être combinés. En particulier, il devrait être possible dans un schéma d'adressage tout à fait général, d'avoir de nombreux niveaux d'indirection : « A2 » pourrait être à nouveau interprétée comme une adresse indirecte, et ainsi de suite.

L'adressage indexé peut lui aussi être combiné avec l'adressage indirect. Cela permet d'accéder efficacement au nième mot d'un bloc de données dès qu'on sait où se trouve le pointeur vers l'adresse de début.

Nous sommes maintenant familiers avec tous les modes d'adressages usuels qui peuvent être fournis dans un système. La plupart des systèmes à microprocesseurs, à cause de la limitation de la complexité du MPU qui doit être réalisé en un seul boîtier, ne fournissent pas tous les modes possibles, mais seulement un sous-ensemble limité de ces modes. Le 6502 fournit un ensemble exceptionnellement riche de possibilités. Examinons le maintenant.

MODES D'ADRESSAGE DU 6502

Adressage implicite (6502)

L'adressage implicite est utilisé par une instruction à un seul octet, qui agit sur les registres internes. Lorsque les instructions implicites agissent exclusivement sur des registres internes, leur exécution prend seulement deux cycles. Lorsqu'elles accèdent à la mémoire, elles prennent trois cycles ou plus. Les instructions qui opèrent uniquement sur les registres internes sont : CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, TYA ainsi que ASL, LSR, ROL et ROR lorsqu'elles agissent sur l'accumulateur.

Les instructions qui demandent des accès-mémoire sont BRK, PHA, PHP, PLA, PLP, RTI et RTS.

Ces instructions ont été décrites au chapitre précédent et leur mode de fonctionnement devrait être clair.

Adressage immédiat (6502)

Comme le 6502 n'a que des registres de travail de 8 bits (PC n'est pas un registre de travail), l'adressage immédiat est, dans le cas du 6502, limité à des constantes 8 bits.

Toutes les instructions ont, par suite, dans le mode d'adressage immédiat, deux octets de long. Le premier octet contient le code opération et le second contient la constante (ou littéral) qui doit être

chargée dans un registre, ou combinée avec un des registres dans une opération arithmétique ou logique. Les instructions qui utilisent ce mode sont : ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, SBC.

Adressage absolu (6502)

Par définition, l'adressage absolu demande trois octets. Le premier octet est le code opération et les deux octets suivants forment l'adresse 16 bits qui spécifie l'emplacement de l'opérande. Sauf dans le cas d'un saut absolu, ce mode requiert quatre cycles.

Les instructions qui peuvent utiliser l'adressage absolu sont : ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, STY.

Adressage en page-zéro (6502)

Par définition, l'adressage page-zéro occupe deux octets : le premier est pour le code opération, le second est pour l'adresse courte, sur 8 bits. L'adressage page-zéro requiert trois cycles. Comme l'adressage page-zéro offre un avantage important en vitesse et en place-mémoire, on doit l'utiliser chaque fois que possible. Ceci exige une gestion précise de la mémoire par le programmeur. Généralement on peut considérer les 256 premiers mots de la mémoire comme l'ensemble des registres de travail du 6502. Toute instruction agit sur ces 256 registres en trois cycles seulement. Cet espace doit donc être soigneusement réservé aux données essentielles auxquelles il faut accéder très rapidement. Les instructions qui peuvent utiliser l'adressage page-zéro sont celles qui utilisent l'adressage absolu sauf JMP et JSR (qui exigent une adresse de 16 bits).

La liste des instructions « légales » est : ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, STY.

Adressage relatif (6502)

Par définition l'adressage relatif utilise deux octets. Le premier est l'instruction de saut tandis que le second spécifie le déplacement et son signe. Afin de différencier ce mode de l'instruction de saut absolu, on appelle ici ces instructions : *branchements*. Dans le cas du 6502, les branchements utilisent toujours l'adressage relatif. Les sauts utilisent toujours l'adressage absolu (plus, bien sûr, les sous-modes qui peuvent lui être combinés, comme l'adressage indirect).

Au point de vue du temps d'exécution, ces instructions doivent être examinées avec précaution. Quand un test est négatif, c'est-à-dire, quand il n'y a pas de branchement, l'instruction prend deux cycles seulement. C'est parce que le compteur ordinal pointe déjà vers la prochaine instruction à exécuter. Mais lorsque le test est positif, c'est-à-dire quand le branchement doit avoir lieu, l'instruction prend trois cycles : il faut calculer une nouvelle adresse effective. La mise à jour du compteur ordinal demande un cycle supplémentaire. Cependant si le branchement franchit une frontière de page, une mise à jour supplémentaire du compteur ordinal est nécessaire et la durée effective de l'instruction devient quatre cycles.

Au point de vue logique, l'utilisateur n'a pas à se soucier de la traversée d'une frontière de page. Le hardware s'en occupe. Mais comme il y a une retenue ou un emprunt supplémentaire à chaque traversée de frontière de page, le temps d'exécution du branchement est changé. Si ce branchement fait partie d'une boucle dont le temps est critique, il faut y faire attention.

Normalement, un bon assembleur indique au programmeur lors de l'assemblage du programme, qu'un branchement franchit une frontière de page, pour le cas où le temps d'exécution serait critique.

Lorsqu'on n'est pas sûr que le branchement va avoir lieu, on doit considérer que, dans certains cas, l'instruction prendra deux cycles, et dans d'autres cas, trois. On utilise souvent un temps moyen.

Les seules instructions qui permettent l'adressage relatif dans le 6502 sont les branchements. Il y a 8 instructions de branchement qui testent les indicateurs du registre d'état pour la valeur « 0 » ou « 1 ». Ce sont : BCC, BCS, BEQ, BMI, BNE, BPL, BVC et BVS.

Adressage indexé (6502)

Le 6502 ne dispose pas d'un adressage indexé tout à fait général, mais le permet d'une manière limitée. Il possède deux registres index. Mais ces registres sont limités à 8 bits. Le contenu d'un registre index est ajouté à la partie adresse de l'instruction. Généralement, le registre index est utilisé comme compteur pour accéder successivement aux éléments d'un bloc ou d'une table. C'est pourquoi il existe des instructions spéciales pour incrémenter ou décrémenter chacun des registres index séparément. En outre, il existe deux instructions spécialisées pour comparer le contenu des registres index à une case-mémoire : cette possibilité de tester les registres index par rapport à une limite est importante pour leur utilisation efficace.

En pratique, la plupart des tables utilisées sont habituellement plus courtes que 256 éléments, et la limitation des registres index à 8 bits est généralement sans conséquence. L'adressage indexé peut

être utilisé, non seulement avec des adresses absolues ordinaires, c'est-à-dire des zones d'adresse de 16 bits, mais aussi avec des adresses page-zéro c'est-à-dire des zones adresses de 8 bits. Il n'y a qu'une restriction. Le registre X peut être utilisé dans les deux cas. Mais le registre Y ne permet que l'adressage *absolu* indexé et non *page-zéro indexé* (sauf pour LDX et STX qui peuvent être modifiées par Y).

L'adressage absolu indexé demande quatre cycles à moins qu'on ne traverse une frontière de page, auquel cas il faut cinq cycles.

Les instructions absolues indexées peuvent faire appel soit à X soit à Y. La liste des instructions qui peuvent utiliser ce mode est :

— avec X : ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA (pas STY) ;

— avec Y : ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC, STA (mais pas ASL, DEC, INC, LSR, ROL, ROR).

En ce qui concerne l'adressage page-zéro indexé, le registre X est le seul autorisé (sauf pour LDX et STX). Les instructions permises sont : ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROR, ROL, SBC, STA, STY.

Adressage indirect (6502)

Le 6502 n'a pas un adressage indirect totalement général. Il restreint la zone adresse à 8 bits. En d'autres termes toutes les adresses indirectes utilisent le sous-mode « page-zéro ». L'adresse effective sur laquelle le code opération va agir est alors formée des 16 bits spécifiés par l'adresse page-zéro située dans l'instruction. En outre, aucune indirection supplémentaire ne peut avoir lieu. Cela veut dire que l'adresse trouvée en page-zéro doit être utilisée telle quelle et non pour une nouvelle indirection.

Enfin, tous les accès indirects doivent être indexés sauf pour JMP.

Toutefois, il faut remarquer que la majorité des microprocesseurs ne possèdent aucun adressage indirect, quel qu'il soit. Enfin, on peut toujours implanter un adressage indirect plus général à l'aide d'une macro.

Il y a deux modes d'adressage indirect possibles : indirect indexé et indexé indirect (sauf pour JMP qui est indirect pur).

Adressage indexé indirect

Dans ce mode, on ajoute le contenu du registre index X à l'adresse page-zéro pour obtenir l'adresse finale sur 16 bits. C'est une manière efficace de retrouver une donnée parmi plusieurs possibles, vers les-

quelles pointent des pointeurs dont le numéro se trouve dans le registre X. Ceci est illustré ci-dessous sur la Figure 5-4.

D'après cette figure, la page-zéro contient une table de pointeurs. Le premier est à l'adresse A qui est dans l'instruction. Si le contenu de X est 2N, alors l'instruction va accéder au pointeur numéro N de la table et retrouver les données vers lesquelles il pointe.

L'adressage indexé indirect demande 6 cycles. Il est naturellement plus lent, que n'importe quel mode direct. Son avantage est la souplesse de programmation ou l'amélioration de vitesse globale qu'il procure.

Les instructions permises sont : ADC, AND, CMP, EOR, LDA, ORA, SBC, STA.

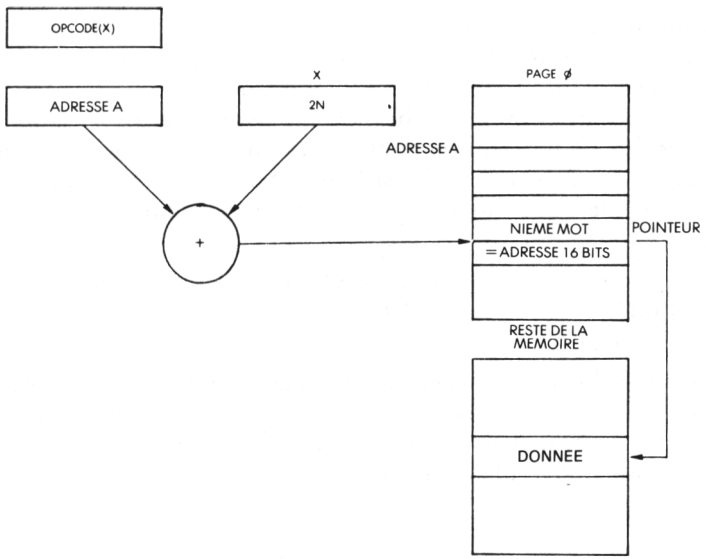


Fig. 5-4 : Adressage indexé indirect

Adressage indirect indexé

Il correspond au mécanisme de post-indexation qui a été décrit dans la section précédente. Là, l'indexation est faite après l'indirection, et non avant. En d'autres termes, l'adresse courte qui figure dans l'instruction sert à accéder à un pointeur 16 bits en page-zéro. Le contenu du registre index Y est alors ajouté comme déplacement à ce pointeur. On accède ensuite à la donnée finale. Dans, ce cas, le

pointeur résidant en page-zéro indique la base d'une table en mémoire. Le registre index Y fournit un déplacement. Il forme un véritable indice à l'intérieur de la table. Cette instruction est particulièrement puissante pour accéder au nième élément d'une table pourvu que l'adresse de début de la table soit sauvegardée en page-zéro. On peut le faire en deux octets seulement. Les instructions permises sont : ADC, AND, CMP, EOR, LDA, ORA, SBC, STA.

UTILISATION DES MODES D'ADRESSAGE DU 6502

Adressage long et court

Nous avons déjà utilisé des instructions de branchement dans les différents programmes que nous avons développés. Leur utilisation devrait être claire. Une question intéressante se pose : que faire si la distance autorisée pour les branchements n'est pas suffisante pour nos besoins ? Une solution simple est d'utiliser ce qu'on appelle un *branchement à distance.* C'est simplement un branchement à un emplacement qui contient une instruction de saut :

```
BCC *+3                 BRANCHEMENT A ADRESSE
                        COURANTE + 3 SI C = 0
JMP LOIN                SINON SAUTE A LOIN
(INSTR SUIVANTE)
```

Les deux lignes de programme ci-dessus entraînent un branchement à l'adresse « LOIN » si la retenue est à un. Ceci résout notre problème de branchement à distance. Considérons maintenant les modes d'adressage plus complexes tels que l'indexation et l'indirection.

Utilisation de l'adressage indexé pour des accès séquentiels dans un bloc

L'indexation est principalement utilisée pour accéder à des emplacements successifs dans une table. La restriction est que le déplacement maximum doit être inférieur à 256 afin de pouvoir faire tenir le déplacement dans un registre index de 8 bits.

Nous avons appris comment reconnaître le caractère « * ». Nous allons maintenant tester dans une table de 100 éléments la présence d'une « * ». L'adresse de départ de cette table est appelée BASE. La table n'a que 100 éléments. Ce nombre est inférieur à 256, et on peut donc utiliser un registre index.

Le programme apparaît ci-dessous :

```
CHERCHE     LDX # 0
SUIVANT     LDA BASE,X
            CMP # '*
            BEQ TROUVE
            INX
            CPX # 100
            BNE SUIVANT
PASDET      .....
TROUVE      .....
```

L'ordinogramme de ce programme apparaît sur la Figure 5-5. Il est conseillé de vérifier que le programme lui est bien équivalent. La logique du programme est simple. Le registre X est utilisé pour pointer vers l'élément considéré de la table. La seconde instruction du programme :

SUIVANT LDA BASE,X

fait appel à l'adressage absolu indexé. Elle spécifie de charger l'accumulateur à partir de l'adresse BASE (adresse absolue sur 16 bits) plus contenu de X. Au départ, le contenu de X est 0. Le premier élément auquel on accède est celui qui se trouve à l'adresse BASE. On peut voir qu'après la prochaine itération, X aura la valeur « 1 » et on accédera à l'élément suivant dans la table à l'adresse BASE + 1.

La troisième instruction du programme : CMP # '* compare la valeur du caractère qui a été lu dans l'accumulateur avec le code de « * ». L'instruction suivante teste le résultat de la comparaison. Si l'égalité a été trouvée, on se branche à l'étiquette TROUVE :

BEQ TROUVE

Sinon on exécute l'instruction à la suite :

INX

Le compteur est augmenté de « 1 ». En se reportant à l'ordinogramme de la Figure 5-5, on constate, en regardant en bas, que la valeur actuelle du registre index doit être testée pour s'assurer qu'on ne sort pas des limites de la table (ici 100 éléments). Ceci est réalisé par l'instruction.

CPX # 100

Cette instruction compare le registre X à la valeur « 100 ». Si le test est négatif, nous devons continuer à chercher le caractère suivant. Ceci est accompli par :

BNE SUIVANT

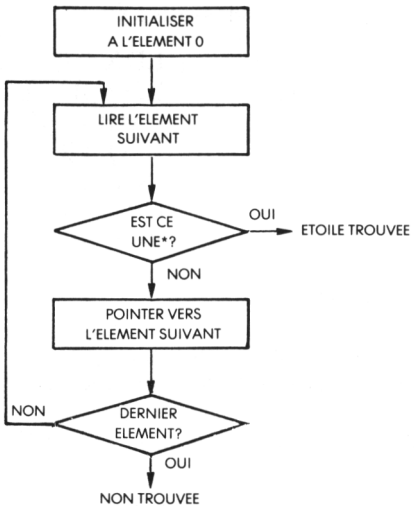


Fig. 5-5 : Recherche d'un caractère dans une table

Cette instruction spécifie un branchement à l'étiquette SUIVANT (seconde instruction du programme), s'il n'y a pas égalité. La boucle sera exécutée tant qu'on n'a pas trouvé une « * » ou que l'index n'a pas atteint la valeur 100. Alors l'instruction qui suit sera exécutée, à l'adresse « PASDET ». Elle correspond au cas où on n'a pas trouvé d'étoile.

Les opérations exécutées selon qu'on a trouvé une « * » ou non n'ont pas d'importance ici ; elles seraient spécifiées par le programmeur. Nous avons maintenant appris à utiliser l'adressage indexé pour accéder à des éléments successifs d'une table. Utilisons cette nouvelle possibilité en augmentant iégèrement la difficulté. Nous allons développer un programme utilitaire important capable de copier un bloc d'une zone mémoire dans une autre. Nous supposerons d'abord que le nombre d'éléments du bloc est inférieur à 256 de façon à pouvoir utiliser le registre index X. Puis nous considérerons le cas général où il est supérieur à 256.

Routine de transfert de bloc de moins de 256 éléments

Nous appellerons « NOMBRE » le nombre d'éléments du bloc à transférer. Il est supposé inférieur à 256. BASE est l'adresse de début du bloc. DEST est l'adresse de base de la zone mémoire d'arrivée. L'algorithme est très simple : on transfère un mot à la fois et on

garde trace du mot qu'on est en train de transférer en stockant sa position dans le registre index X. Le programme apparaît ci-dessous :

```
            LDX   #NOMBRE
SUIVANT     LDA   BASE, X
            STA   DEST, X
            DEX
            BNE   SUIVANT
```

Examinons-le :

LDX #NOMBRE

Cette ligne du programme charge le nombre N de mots à transférer dans le registre index. L'instruction suivante charge le mot numéro N du bloc dans l'accumulateur et la troisième le dépose dans la zone d'arrivée (voir Figure 5-6).

ATTENTION : ce programme ne fonctionnera correctement que si l'adresse BASE est supposée pointer juste *en dessous* du bloc, de même pour l'adresse DEST. Sinon il faut un ajustement du programme.

Lorsqu'un mot a été transféré de la zone de départ à la zone d'arrivée, il faut mettre à jour le registre index. Ceci est effectué par l'instruction DEX, qui le décrémente. Ensuite, le programme teste simplement si X a décrémenté à « 0 ». Si oui, le programme est terminé. Sinon, on boucle en retournant à l'étiquette « SUIVANT ».

Vous remarquerez que quand X = 0, le programme *ne boucle pas.* Par suite le mot situé à l'adresse BASE n'est pas transféré. Le dernier mot transféré est celui de l'adresse BASE + 1. C'est pourquoi nous avons supposé que la base pointait juste *en dessous* du bloc.

Exercice 5.1 : *Modifier le programme ci-dessus en supposant que BASE et DEST pointent sur le premier élément du bloc.*

Ce programme illustre aussi l'utilisation des compteurs de boucle. Vous noterez que X a été chargé avec la *valeur finale,* puis *décrémenté* et testé.

Au premier abord, il pourrait sembler plus simple de commencer avec « 0 » dans X puis de l'incrémenter jusqu'à ce qu'il atteigne la valeur maximum. Mais pour tester si X a atteint son maximum, il faudrait une instruction supplémentaire (l'instruction de comparaison). La boucle ferait alors 5 instructions au lieu de 4. Comme ce programme de transfert sera utilisé normalement pour de grands

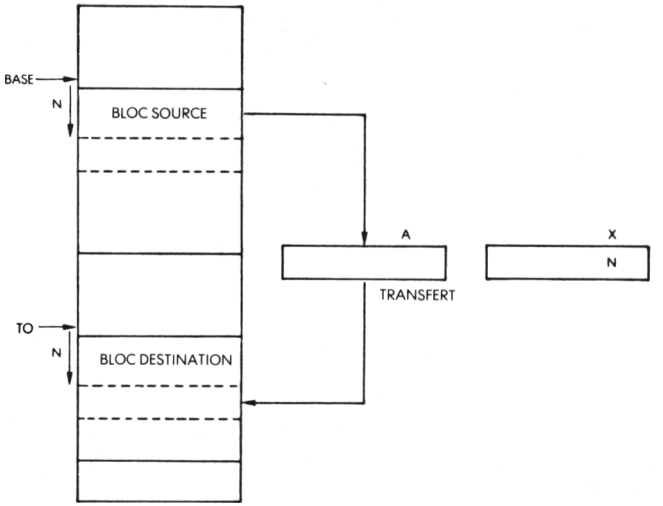


Fig. 5-6 : Organisation de la mémoire pour le transfert de bloc

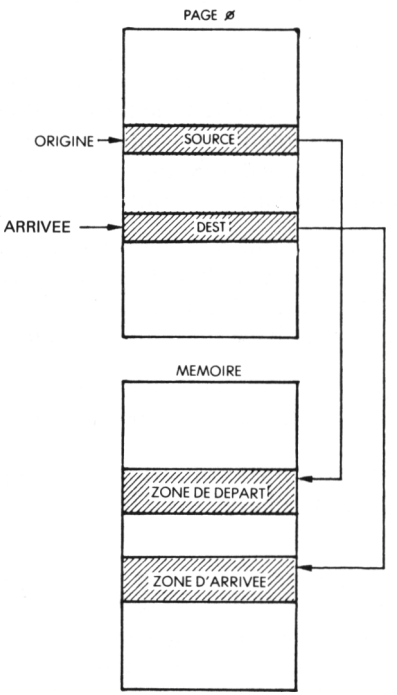


Fig. 5-7 : Plan de la mémoire pour le transfert de bloc général

nombres de mots, il est important de réduire le nombre d'instructions de la boucle. C'est pourquoi, au moins pour les courtes boucles, le registre index est de préférence *décrémenté* qu'*incrémenté*.

Routine de transfert de bloc (plus de 256 éléments)

Considérons maintenant le cas général du transfert d'un bloc qui peut comporter plus de 256 éléments. Nous ne pouvons plus utiliser un seul registre index puisque 8 bits ne suffisent pas pour stocker un nombre supérieur à 256. L'organisation de la mémoire relative à ce programme est montrée sur la Figure 5-7. La longueur du bloc mémoire à transférer nécessite 16 bits, et par suite, est rangée en mémoire.

La partie haute représente le nombre de blocs de 256 mots « BLOCS ». Le reste est appelé « RESTE », et il représente le nombre de mots à transférer lorsque tous les blocs de 256 mots ont été transférés. Les adresses de départ et d'arrivée seront en mémoire aux emplacements DEP et ARR. Supposons d'abord que RESTE est nul, c'est-à-dire qu'on transfère un nombre entier de blocs de 256 mots. Le programme apparaît ci-dessous :

```
          LDA #SOURCEBAS
          STA DEP
          LDA #SOURCEHAUT
          STA DEP + 1          RANGEMENT ADRESSE
          LDA #DESTBAS         DE DEPART
          STA ARR
          LDA #DESTHAUT        RANGEMENT ADRESSE
          STA ARR + 1          D'ARRIVEE
          LDX #BLOCS           NOMBRE DE BLOCS
          LDY #0               TAILLE DE BLOC
SUIVANT   LDA (DEP),Y          LIRE UN ELEMENT
          STA (ARR),Y          LE TRANSFERER
          DEY                  MISE A JOUR
          BNE SUIVANT          FINI ?
BLOCSUI   INC DEP + 1          INCREMENTER
          INC ARR + 1          POINTEUR DE BLOC
          DEX                  COMPTEUR DE BLOCS
          BMI FIN
          BNE SUIVANT
          LDY #RESTE
          BNE SUIVANT
```

L'adresse de départ 16-bits est rangée à l'adresse « DEP » par les quatre premières instructions. Les quatre suivantes font de même pour l'adresse d'arrivée qui est rangée en « ARR ». Comme nous avons à transférer un nombre de mots plus grand que 256, nous utilisons, simplement deux registres de 8 bits. L'instruction suivante charge le registre X avec le nombre de blocs à transférer. C'est la 9e instruction du programme. La suivante met la valeur zéro dans le registre index Y pour le préparer au transfert de 256 mots. Nous allons utiliser l'adressage indirect indexé. Il faut se rappeler qu'il met en jeu d'abord une indirection en page-zéro puis un accès indexé à l'adresse 16 bits obtenue grâce au registre index. Regardons le programme :

```
SUIVANT    LDA    (DEP),Y
```

Cette instruction charge l'accumulateur avec le contenu de la case-mémoire dont l'adresse est départ plus contenu du registre index Y. Cf. Figure 5-7 pour la carte d'implantation mémoire. Ici, le contenu de Y est initialement 0. « A » sera donc chargé avec le contenu de la case-mémoire d'adresse « SOURCE ». Notez que ici, à la différence de l'exemple précédent, nous supposons que SOURCE est l'adresse du premier mot à l'intérieur du bloc.

A l'aide de la même technique, l'instruction suivante déposera le contenu de l'accumulateur (le premier mot du bloc à transférer) à l'emplacement d'arrivée approprié STA (ARR),Y.

Exactement comme dans le cas précédent, nous décrémentons le registre index pour boucler 256 fois. Ceci est réalisé par les deux instructions qui suivent :

```
DEY
BNE    SUIVANT
```

ATTENTION : on utilise ici une astuce pour diminuer la taille du programme. Le lecteur attentif a noté que le registre Y est *décrémenté*. Le premier mot à transmettre est celui de la position « 0 ». Le suivant sera le mot N° 255. Ceci est dû au fait que, si l'on décrémente « 0 », on obtient huit « 1 » dans le registre (soit 255). Le lecteur s'assurera qu'il n'y a pas d'erreur. Quand le registre Y est décrémenté pour arriver à « 0 », il n'y a *pas* de transfert. La prochaine instruction exécutée sera BLOCSUI... Donc on aura transféré exactement 256 mots. Il est clair qu'on aurait pu utiliser cette astuce dans le programme précédent pour le raccourcir.

Lorsqu'un bloc complet a été transféré, il n'y a plus qu'à pointer vers la page suivante dans notre bloc de départ et notre bloc d'arrivée. Ceci est accompli en ajoutant « 1 » à la partie haute de l'adresse

pour la source et la destination. C'est ce que font les deux instructions suivantes du programme :

```
BLOCSUI     INC DEP + 1
            INC ARR + 1
```

Après avoir incrémenté le pointeur de page, on teste s'il y a un autre bloc à transférer ou non, en décrémentant le compteur de blocs implanté en X. Ceci est accompli par :

```
DEX
```

Si tous les blocs ont été transférés, on sort du programme en branchant à l'étiquette FIN :

```
BMI  FIN
```

Sinon, il y a deux possibilités : ou bien nous n'avons pas décrémenté jusqu'à « 0 », ou bien nous sommes arrivés exactement à « 0 ». Si l'on n'est pas encore à « 0 », on branche à SUIVANT :

```
BNE SUIVANT
```

Si l'on est exactement à « 0 », c'est qu'il faut encore transférer les mots spécifiés par « RESTE ». C'est la dernière partie de notre transfert. Ceci est accompli par :

```
LDY    # RESTE
```

qui charge l'index Y avec le compte des mots qui restent à transférer.

On se branche alors à « SUIVANT » :

```
BNE SUIVANT
```

Le lecteur s'assurera que, lors de cette dernière boucle, soit on va se brancher vers SUIVANT, la prochaine fois qu'on reviendra à BLOCSUI, soit on va effectivement sortir pour de bon du programme. Ceci est dû au fait que l'index X à la valeur « 0 » avant d'arriver à BLOCSUI. La troisième instruction après BLOCSUI le ramène à « — 1 » et on sortira à FIN.

Addition de deux blocs

Cet exemple illustrera de manière simple l'utilisation d'un registre index pour l'addition de deux blocs de moins de 256 éléments. Ensuite le programme suivant utilisera l'adressage indirect indexé pour accéder à des blocs dont l'adresse réside à un endroit connu,

mais qui est elle-même inconnue. Le programme apparaît ci-dessous :

```
ADBLOC  LDY  #NBR-1      CHARGER LE COMPTEUR
SUIVANT CLC
                         LIRE LES ELEMENTS
        LDA  PTR1,Y      SUIVANTS
        ADC  PTR2,Y      LES AJOUTER
        STA  PTR3,Y      RANGER LE RESULTAT
                         DECREMENTER
        DEY              LE COMPTEUR
        BPL  SUIVANT     FINI ?
```

L'index Y est utilisé comme compteur et est chargé avec le nombre d'éléments moins un. Nous supposons que le pointeur PTR1 pointe vers le premier élément du BLOC1, PTR2 vers le premier élément du BLOC2, et PTR3 pointe vers la zone destination où doivent être stockés les résultats.

Le programme devrait être clair. Le dernier élément de BLOC1 est lu dans l'accumulateur, puis ajouté au dernier élément de BLOC2. Il est ensuite rangé à l'emplacement voulu de BLOC3. Les éléments suivants sont ajoutés, et ainsi de suite.

Même exercice avec l'adressage indirect indexé

Nous supposons maintenant que les adresses PTR1, PTR2 et PTR3 ne sont pas connues a priori. Mais nous savons qu'elles sont rangées en page-zéro, aux adresses LOC1, LOC2 et LOC3. Ceci est un mécanisme usuel pour passer des informations entre sous-programmes. Le programme correspondant apparaît ci-dessous :

```
ADBLOC  LDY  #NBR-1
SUIVANT CLC
        LDA  (LOC1),Y
        ADC  (LOC2),Y
        STA  (LOC3),Y
        DEY
        BPL  SUIVANT
```

La correspondance entre ce programme et le précédent devrait maintenant être évidente. Il illustre clairement l'intérêt de l'adressage indirect indexé quand l'adresse absolue n'est pas connue au moment où on écrit le programme, mais l'emplacement de cette adresse est connu. On peut noter que les deux programmes ont exactement le même nombre d'instructions. Un exercice intéressant serait maintenant de déterminer lequel des deux est le plus rapide à l'exécution.

Exercice 5.2 : *Calculer, à l'aide des tables de l'appendice, le nombre d'octets et le nombre de cycles correspondant à ces deux programmes.*

RECAPITULATION

Une description complète des modes d'adressage a été présentée. Il a été démontré que le 6502 offre la majorité des mécanismes d'adressage possibles, et ses caractéristiques ont été analysées. Enfin plusieurs programmes d'application ont été présentés pour démontrer le rôle de chacun des modes d'adressage. La programmation du 6502 exige la compréhension de ces mécanismes.

EXERCICES

5.3 : *Ecrire un programme qui ajoute les 10 premiers octets d'une table située à l'adresse « BASE ». Le résultat aura 16 bits (c'est le calcul d'une somme de contrôle).*

5.4 : *Pouvez-vous traiter le même problème sans utiliser l'adressage indexé ?*

5.5 : *Inversez l'ordre des 10 octets de cette table. Rangez le résultat à l'adresse « INVER ».*

5.6 : *Cherchez, dans la même table, l'élément le plus grand. Rangez le dans la case-mémoire « MAX ».*

5.7 : *Ajoutez ensemble les éléments correspondants de trois tables d'adresses de base BASE1, BASE2, BASE3. La longueur de ces tables est rangée en page-zéro à l'adresse « LONG ».*

6

TECHNIQUES D'ENTREES-SORTIES

INTRODUCTION

Nous avons appris jusqu'ici comment échanger des informations entre la mémoire et les différents registres du processeur. Nous avons appris à gérer les registres et à utiliser diverses instructions pour manipuler les données. Nous devons maintenant apprendre à communiquer avec le monde extérieur. Ce sont les « entrées-sorties ».

Les entrées désignent la saisie de données à partir de périphériques extérieurs (clavier, disque, capteur de grandeur physique). *Les sorties* désignent les transferts de données depuis le processeur ou la mémoire vers des unités extérieures telles qu'une imprimante, un écran, des actionneurs ou des relais.

Nous allons procéder en deux étapes. D'abord nous apprendrons à effectuer les opérations d'entrées-sorties requises par les périphériques usuels. Ensuite nous apprendrons à gérer plusieurs unités d'entrées-sorties simultanément, c'est-à-dire à les *synchroniser*. Cette seconde partie couvrira en particulier le choix entre *scrutation* ou *interruptions*.

ENTREES-SORTIES

Dans cette section, nous apprendrons à capter ou à générer des signaux simples tels que des impulsions. Puis, nous étudierons les techniques pour forcer ou mesurer des délais. Nous serons alors prêts à accomplir des entrées-sorties plus complexes comme les transferts rapides en série ou en parallèle.

Génération d'un signal

Dans le cas le plus simple, un périphérique de sortie sera mis en marche ou arrêté par le processeur. Pour changer l'état du périphérique, le programmeur va simplement changer un niveau logique de « 1 » à « 0 » ou de « 0 » à « 1 ».

Supposons que le bit 0 d'un registre appelé « SORTIE1 » soit relié à un relais extérieur. Pour le mettre en marche nous allons simplement écrire un « 1 » dans le bit voulu du registre. Nous supposons que SORTIE1 représente l'adresse de ce registre de sortie dans notre système. Le programme qui met en marche le relais est :

```
MARCHE LDA  =% 00000001
       STA  SORTIE1
```

Nous avons supposé que l'état des 7 autres bits du registre SORTIE1 est sans signification. Mais ce n'est souvent pas le cas. Ces bits pourraient être reliés à d'autres relais. Améliorons donc ce programme simple. Nous voulons mettre le relais en marche sans changer l'état d'aucun autre bit du registre. Nous supposons que le registre peut être lu ou écrit.

Notre programme devient alors :

```
MARCHE  LDA SORTIE1      LIRE LE CONTENU DE SORTIE
        ORA #%00000001   FORCER A « 1 » le BIT 0
        STA SORTIE1
```

Le programme lit d'abord le contenu de l'adresse SORTIE 1 puis effectue un ou inclusif sur son contenu. Ceci force à « 1 » le bit en position 0 et laisse intact le reste du registre (pour plus de détails sur l'instruction ORA, cf. Chapitre 4). Ceci est illustré par la Figure 6-1.

Impulsions

On génère une impulsion exactement comme dans le cas d'un *niveau,* vu ci-dessus. On met un bit de sortie d'abord à « 1 », puis à « 0 ». Il en résulte une impulsion. Ceci est illustré sur la Figure 6-2. Néanmoins, il y a, cette fois, un problème de plus à résoudre : on doit générer une impulsion de durée correcte. Etudions donc comment générer un délai calculé.

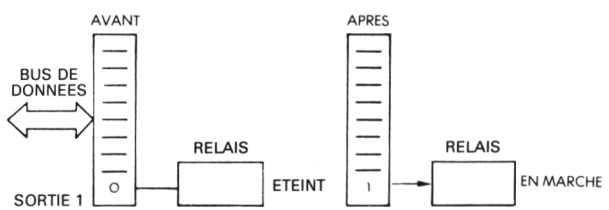


Fig. 6-1 : Mise en marche d'un relais

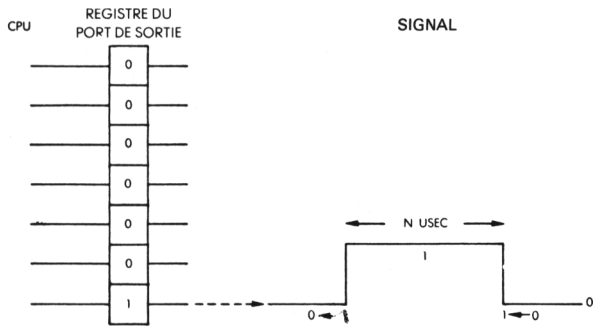


Fig. 6-2 : Impulsion programmée

Génération et mesure d'un délai

Un délai peut être généré soit par des méthodes software (logiciel), soit par hardware (matériel). Nous verrons ici comment le faire par programme, et plus tard, comment cela peut être accompli avec un compteur hardware appelé générateur d'intervalles de temps programmable (temporisateur).

Les délais programmés sont obtenus par comptage. On charge un registre de comptage avec une valeur, puis on décrémente. Le programme boucle sur lui-même et continue à décrémenter jusqu'à ce que le compteur atteigne la valeur « 0 ». Le temps total requis par ce processus va réaliser le délai souhaité. Par exemple, réalisons un délai de 36 microsecondes :

```
DELAI   LDY#07      LE COMPTEUR EST Y
ENCORE  DEY         DECREMENTER
        BNE ENCORE  TEST
```

Ce programme met la valeur 7 dans le registre index Y. L'instruction suivante décrémente Y et la suivante entraîne un branchement à ENCORE, tant que Y n'a pas décrémenté jusqu'à « 0 ».

Quand Y atteint finalement « 0 », le programme sort de la boucle et exécute les instructions qui peuvent suivre. La logique de ce programme est simple et elle est conforme à l'ordinogramme de la Figure 6-3.

Calculons maintenant le délai effectif qui sera réalisé par ce programme. Regardons dans l'appendice de la fin du livre le nombre de cycles correspondant à chacune des instructions :

LDY, en mode immédiat, demande 2 cycles. DEY prend 2 cycles. Enfin BNE prend 3 cycles. En regardant le nombre de cycles demandés par BNE dans la table, on vérifie qu'il y a 3 possibilités : si le

branchement n'a pas lieu, BNE ne demande que 2 cycles. Si le branchement a lieu, ce qui est le cas normal pendant la boucle, il faut un cycle de plus. Enfin, si on traverse une frontière de page, il faut un cycle supplémentaire. On supposera ici qu'on ne franchit pas de frontière de page.

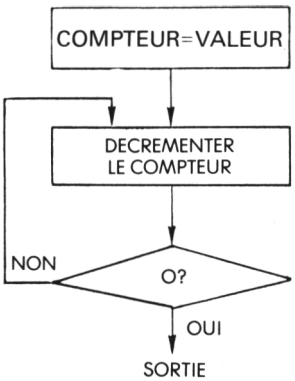


Fig. 6-3 : Ordinogramme d'un délai

La durée est donc 2 cycles pour la première instruction, plus 5 cycles pour les deux suivantes multiplié par le nombre de fois que la boucle est exécutée, moins 1 cycle car dans le dernier BNE le branchement n'a pas lieu :

Délai = $2 + 5 \times 7 - 1 = 36$

Si l'on suppose un temps de cycle de 1 microseconde, ce délai programmé sera donc de 36 microsecondes.

On peut voir sur cet exemple simple que la résolution la plus fine avec laquelle on peut ajuster ce délai est de 5 microsecondes. Le délai minimum est 6 microsecondes.

Exercice 6.1 : *Quel est le délai maximum qu'on peut réaliser avec ces trois instructions ? Pouvez-vous modifier le programme pour obtenir un délai d'une microseconde ?*

Exercice 6.2 : *Modifier le programme pour obtenir un délai d'environ 100 microsecondes.*

Si l'on souhaite réaliser un délai plus long, une solution simple est d'ajouter des instructions dans le programme entre DEY et BNE. La manière la plus simple de le faire est d'inclure des instructions NOP (l'instruction NOP ne fait rien, mais dure 2 cycles).

Délais plus longs

La génération software de délais plus longs peut se faire en utilisant un compteur plus large. On peut utiliser deux registres internes,

ou mieux deux mots-mémoire, pour maintenir un compteur sur 16 bits. Pour simplifier, supposons que la partie basse du compte soit « 0 ». L'octet inférieur sera chargé avec « 255 » (le compte maximum), puis il entrera dans une boucle de décrémentation. A chaque fois qu'il sera décrémenté à « 0 », l'octet supérieur du compteur sera décrémenté. Quand l'octet supérieur atteindra la valeur « 0 », le programme s'arrêtera. Si l'on a besoin de plus de précision dans le délai à générer, la partie basse du compte peut avoir une valeur non nulle. Dans ce cas, on écrirait le programme de trois lignes qui a été décrit ci-dessus. Naturellement, on pourrait générer des délais encore plus longs en utilisant plus de deux mots. Ceci est analogue au fonctionnement d'un indicateur kilométrique sur une voiture. Quand la roulette la plus à droite passe de « 9 » à « 0 », la roulette qui est à sa gauche est incrémentée de 1. C'est le principe général du comptage avec des unités discrètes.

Cependant, la principale objection est que, lorsqu'on décompte de longs délais, le microprocesseur ne fait rien d'autre pendant des centaines de millisecondes ou même des secondes. Si l'ordinateur n'a rien d'autre à faire, ceci est parfaitement acceptable. Néanmoins, dans le cas général, il faut que le microordinateur reste disponible pour d'autres tâches, de sorte que l'on ne réalise normalement pas les longs délais par software. En fait, même de courts délais programmés peuvent être critiquables dans un système, s'il faut fournir des temps de réponse garantis dans certaines situations. On doit alors réaliser le délai par hardware. En outre, si l'on utilise les interruptions, l'exactitude des délais risque d'être perdue si la boucle de comptage peut être interrompue.

Exercice 6.3 : *Ecrire un programme qui réalise un délai de 100 ms (pour un télétype).*

Délais hardware

Les délais sont réalisés par hardware à l'aide d'un générateur d'intervalles de temps programmable ou, brièvement, d'un « temporisateur ».

Un registre du temporisateur est chargé avec une valeur. La différence est que, cette fois, le temporisateur va automatiquement décrémenter ce compteur, périodiquement. La période est généralement ajustable ou sélectionnable par le programmeur. Quand le temporisateur aura décrémenté jusqu'à « 0 », il enverra normalement une interruption au microprocesseur. Il peut aussi positionner un bit d'état qui peut être testé périodiquement par l'ordinateur. L'utilisation des interruptions sera expliquée plus loin dans ce chapitre.

D'autres modes de fonctionnement du temporisateur peuvent permettre de partir de « 0 » et de mesurer la durée d'un signal, ou

encore de compter le nombre d'impulsions reçues. Lorsqu'il fonctionne en générateur d'intervalles, le temporisateur est dit fonctionner en *monostable.* Lorsqu'il compte des impulsions, il est dit fonctionner en mode *compteur.* Certains boîtiers temporisateurs peuvent même comprendre des registres multiples et un certain nombre d'options qui peuvent être sélectionnées par programme. C'est le cas, par exemple, des temporisateurs contenus dans le 6522, boîtier décrit au chapitre suivant.

Détection d'impulsions

La détection d'impulsions constitue le problème inverse de la génération d'impulsions, avec une difficulté en plus : alors que les impulsions de sortie sont générées sous le contrôle du programme, les impulsions d'entrée arrivent de façon *asynchrone* par rapport au programme. Pour détecter une impulsion, on peut faire appel à deux méthodes : *scrutation* et *interruptions.* Les interruptions seront discutées plus loin dans ce chapitre.

Considérons maintenant la technique de scrutation. Dans cette technique, le programme lit continuellement la valeur d'un registre d'entrée donné, en testant une position, par exemple le bit 0. On supposera que le bit 0 est « 0 » au départ. Quand on reçoit une impulsion, ce bit passe à « 1 ». Le programme surveille constamment le bit 0 jusqu'à ce qu'il prenne la valeur « 1 ». Quand on détecte un « 1 », on a détecté l'impulsion.

Le programme apparaît ci-dessous :

```
SCRUTE    LDA #$01
ENCORE    BIT ENTREE
          BEQ ENCORE
SUITE     .....
```

Inversèment, supposons que la ligne d'entrée soit normalement à « 1 » et qu'on souhaite détecter un « 0 ». C'est le cas par exemple pour détecter un bit « START » (de départ), lorsqu'on surveille une ligne reliée à un télétype. Le programme apparaît ci-dessous :

```
SCRUTE    LDA #$01
SUIVANT   BIT ENTREE
          BNE SUIVANT
DEPART    .....
```

Mesure de la durée

La mesure de la durée de l'impulsion peut se faire de la même façon que pour calculer la durée d'une impulsion en sortie. On peut faire appel soit à une technique hardware soit à une technique software. Quand on mesure une impulsion par software, on incrémente

un compteur régulièrement, puis on vérifie la présence de l'impulsion. Si l'impulsion est toujours présente, le programme reboucle sur lui-même. Quand l'impulsion disparaît, on utilise le compte contenu dans le registre compteur pour calculer la durée effective de l'impulsion. Le programme apparaît ci-dessous :

```
DUREE   LDX #0          METTRE LE COMPTEUR A ZERO
        LDA #$01        ON SURVEILLE LE BIT 0
ATTEND  BIT ENTREE
        BEQ ATTEND
ENCORE  INX
        BIT ENTREE
        BNE ENCORE
```

Naturellement, on suppose que la durée maximum de l'impulsion ne va pas entraîner de dépassement de capacité du registre X. Si c'était le cas, il faudrait allonger le programme pour en tenir compte (ou alors ce serait une erreur de programmation). Comme nous savons maintenant comment détecter et générer des impulsions, nous allons acquérir ou transférer de plus grandes quantités de données.

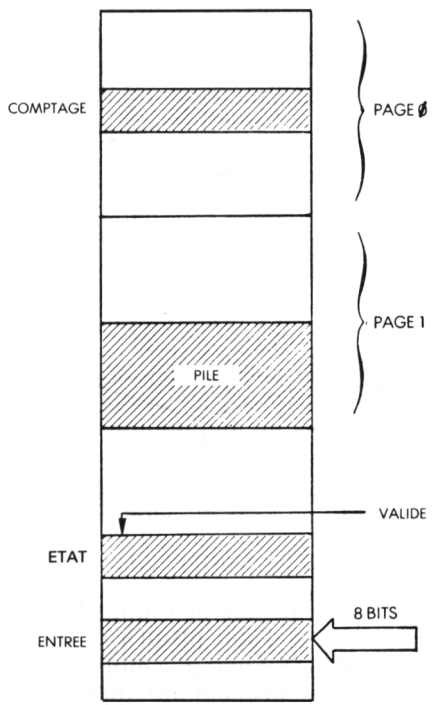


Fig. 6-4 : Transfert de mots en parallèles : la mémoire

On distinguera deux cas : données série et données parallèles. Ensuite nous appliquerons effectivement ces connaissances à des périphériques existants.

TRANSFERT DE MOTS EN PARALLELE

On suppose ici que les 8 bits de données à transférer sont disponibles en parallèle à l'adresse « ENTREE ». Le microprocesseur doit lire le mot de données à cette adresse chaque fois qu'un mot d'état indique qu'il est valide. On supposera que l'information d'état est contenue dans le bit 7 de l'adresse « ETAT ». Nous allons écrire un programme qui lira et sauvegardera automatiquement chaque mot de données tel qu'il arrive. Pour simplifier, nous supposerons que le nombre de mots à lire est connu à l'avance et est contenu à l'adresse « COMPTE ». Si cette information n'était pas disponible on testerait par rapport à un caractère dit *caractère d'arrêt* comme un *effacement* ou, peut être, une « * ». Nous avons déjà appris à le faire.

SCRUTATION OU DEMANDE DE SERVICE

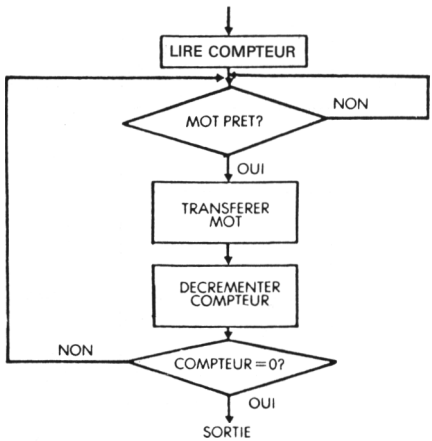


Fig. 6-5 : Transfert de mots en parallèles : ordinogramme

L'ordinogramme apparaît sur la Figure 6-5. Il est très simple. Nous testons l'information d'état jusqu'à ce qu'elle devienne « 1 », signalant qu'un mot est prêt. Quand le mot est prêt, nous le lisons et le sauvons à un emplacement-mémoire approprié. Nous décrémentons ensuite le compte et testons s'il a atteint la valeur « 0 ». Si oui,

nous avons fini, sinon, nous lisons le mot suivant. Le programme qui suit cet algorithme apparaît ci-dessous :

```
PARAL   LDX  COMPTE   COMPTEUR
ATTEND  LDA  ETAT     LE BIT 7 EST A
                      1 SI LA DONNEE EST VALIDE
        BPL  ATTEND   DONNEE VALIDE ?
        LDA  ENTREE   LIRE LA DONNEE
        PHA           LA SAUVER DANS LA PILE
        DEX
        BNE  ATTEND
```

Les deux premières instructions du programme lisent l'information d'état et entraînent qu'on boucle tant que le bit 7 du registre d'état est « 0 ». (C'est le bit de signe c'est-à-dire le bit N).

```
ATTEND  LDA  ETAT
        BPL  ATTEND
```

Quand BPL est négatif, la donnée est valide et on peut lire :

```
        LDA  ENTREE
```

Le mot a maintenant été lu à partir de l'adresse ENTREE où il se trouvait, et il faut le sauver. Supposant que le nombre de mots à transférer est assez petit, on utilise un

```
        PHA
```

Si la pile risque d'être pleine, ou si le nombre de mots à transférer est grand, on ne pourrait pas les mettre dans la pile et il faudrait les ranger dans une zone mémoire donnée, à l'aide, par exemple, d'une instruction indexée. Toutefois ceci exigerait une instruction de plus pour incrémenter ou décrémenter le registre index. PHA est plus rapide.

Le mot de donnée a maintenant été lu et sauvegardé. Nous décrémentons simplement le compteur et testons si c'est fini.

```
        DEX
        BNE  ATTEND
```

Nous continuons à boucler jusqu'à ce que le compteur arrive à « 0 ». Ce programme de 6 instructions peut être considéré comme « benchmark », c'est-à-dire programme de mesure de performances. Un benchmark est un programme optimisé avec soin destiné à tester les capacités d'un processeur donné dans une situation spécifique. Les transferts parallèles constituent une telle situation. Ce programme a été conçu en vue d'une efficacité et d'une vitesse maxi-

mum. Calculons maintenant la vitesse de transfert maximum permise par ce programme. Nous supposerons que le compte est en page 0. La durée de chaque instruction s'obtient en examinant la table à la fin du livre. On trouve :

```
                          CYCLES
        LDX  COMPTE       3
ATTEND  LDA  ETAT         4
        BPL  ATTEND       2/3     (va en
        LDA  ENTREE       4       séquence/branchement)
        PHA               3
        DEX               2
        BNE  ATTEND       2/3
```

Le temps d'exécution minimum s'obtient en supposant que la donnée est toujours disponible quand on teste l'état. En d'autres termes le premier BPL est supposé conduire en séquence à chaque fois. Le temps est alors :

$3 + (4 + 2 + 4 + 3 + 2 + 3) \times \text{COMPTE}$

Si on néglige les 3 premières microsecondes nécessaires pour initialiser le registre de comptage, le temps passé à transférer un mot est de 18 microsecondes. Le taux de transfert maximum est donc :

$$\frac{1}{18.10^{-6}} = 55 \text{ K octets par seconde}$$

Exercice 6.4 : *Supposez que le nombre de mots à transférer soit supérieur à 256. Modifier le programme en conséquence et déterminez l'impact sur le taux de transfert maximum.*

Nous avons maintenant appris à effectuer des transferts parallèles rapides. Considérons un cas plus complexe.

TRANSFERT SERIE

Une entrée série est une entrée où les bits d'information (« 0 » ou « 1 ») arrivent successivement sur une ligne. Ces bits peuvent arriver à intervalles réguliers. Ceci est alors appelé transmission *synchrone*. Ou bien, ils peuvent arriver en groupes à des instants aléatoires. C'est ce qu'on appelle transmission *asynchrone*. Nous allons développer un programme qui peut travailler dans les deux cas. Le principe de la saisie de données séquentielles est simple : nous surveillerons une ligne d'entrée, qu'on supposera être la ligne « 0 ». Quand un bit de données sera détecté sur cette ligne, nous le lirons et le ferons entrer par décalage dans un registre de mémorisation. Quand 8 bits

auront été assemblés, nous préserverons l'octet de donnée en mémoire et nous assemblerons le suivant.

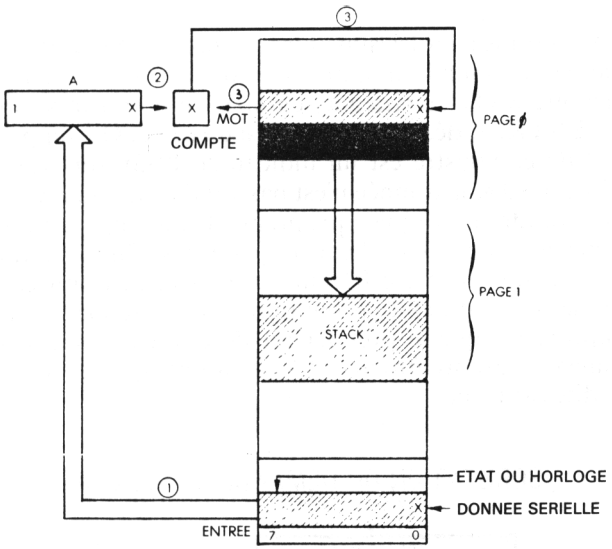


Fig. 6-6 : Conversion série-parallèle

Pour simplifier, nous supposerons connu à l'avance le nombre d'octets à recevoir. Sinon, nous pourrions par exemple, tester l'arrivée d'un caractère spécial d'arrêt et arrêter le transfert à ce moment. Nous avons appris à le faire. L'ordinogramme de ce programme apparaît sur la Figure 6-7. Le programme apparaît ci-dessous :

```
SERIE   LDA #$00
        STA  MOT
BOUCLE  LDA  ENTREE  LE BIT 7 EST L'ETAT,
                     LE BIT 0 EST LA DONNEE
        BPL  BOUCLE  BIT REÇU ?
        LSR  A       LE DECALER DANS LA RETENUE
        ROL  MOT     LE SAUVER EN MEMOIRE
        BCC  BOUCLE  CONTINUER SI RETENUE = 0
        LDA  MOT
        PHA          SAUVEGARDE DU MOT
        LDA  #$01    INITIALISER COMPTEUR DE BITS
        STA  MOT
        DEC  COMPTE  DECREMENTER COMPTEUR
                     DE MOTS
        BNE  BOUCLE  ASSEMBLER LE MOT SUIVANT
```

Ce programme a été conçu pour être efficace grâce à des techniques nouvelles que nous allons maintenant expliquer (cf. Figure 6-6).

Les conventions sont les suivantes : l'adresse-mémoire COMPTE est supposée contenir le nombre de mots à transférer. L'emplacement-mémoire MOT servira à assembler 8 bits entrants consécutifs. L'adresse ENTREE désigne un registre d'entrée. On suppose que la position 7 de ce registre est un indicateur d'état, ou bit d'horloge. Quand il vaut « 0 », la donnée n'est pas valide. Quand il vaut « 1 », la donnée est valide. La donnée proprement dite apparaît au bit 0 de cette même adresse. Dans beaucoup de cas, l'information d'état apparaît dans un registre différent du registre de données. Il serait alors simple de modifier le programme en conséquence. En outre, nous supposerons que le premier bit de données à recevoir par ce programme est garanti être un « 1 ». Il indique que les données proprement dites suivent.

SCRUTATION OU DEMANDE DE SERVICE

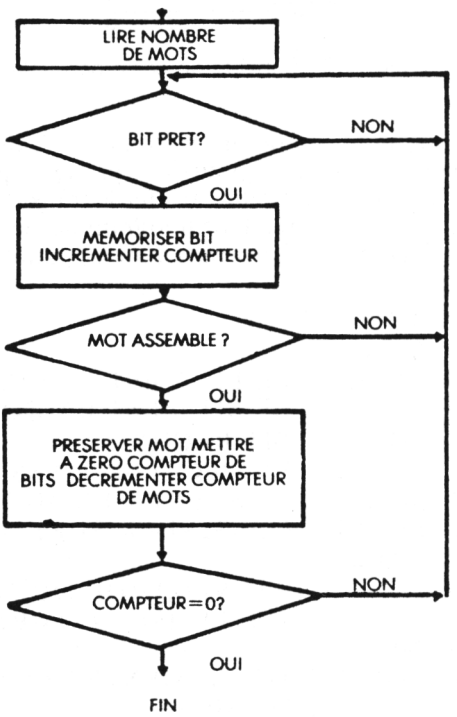


Fig. 6-7 : Transfert série : ordinogramme

Si ce n'était pas le cas, nous verrons plus tard une modification évidente pour en tenir compte. Le programme correspond exactement à l'ordinogramme de la Figure 6-7. Les quelques premières lignes du programme forment une boucle d'attente qui teste si un bit est prêt. Pour le déterminer, nous lisons le registre d'entrée puis testons le bit de signe (N). Tant que ce bit est « 0 », l'instruction BPL réussit et on retourne à BOUCLE. Quand le bit d'état (ou d'horloge) deviendra « 1 », le BPL ne causera plus de branchement, et on exécutera l'instruction qui suit.

Rappelez-vous que BPL veut dire « branchement si plus », c'est-à-dire quand le bit 7 (bit de signe) est « 0 ». Cette séquence d'instructions correspond à la flèche 1 de la Figure 6-6.

A ce moment, l'accumulateur contient un « 1 » en position 7 et le bit de donnée proprement dite est en position 0. Le premier bit de donnée sera un « 1 ». Mais les suivants peuvent être soit « 0 » soit « 1 ». Nous souhaitons maintenant sauvegarder le bit de donnée qui a été recueilli en position 0. L'instruction :

LSR A

décale d'un cran à droite le contenu de l'accumulateur. Ceci amène le bit le plus à droite de A, c'est-à-dire notre bit de donnée, dans la retenue. Nous allons maintenant le préserver dans la case-mémoire MOT (ceci est illustré par les flèches 2 et 3 de la Figure 6-6).

ROL MOT

L'effet de cette instruction est de transférer le bit de retenue dans la position la plus à droite de l'adresse MOT. En même temps, le bit le plus à gauche de MOT tombe dans l'indicateur de retenue. (Si vous avez des doutes sur le fonctionnement de la rotation, reportez-vous au Chapitre 4). Il est important de se rappeler qu'une rotation a pour effet, à la fois de sauver le bit de retenue dans la position la plus à droite, et de recopier la valeur du bit 7 dans le bit de retenue.

Ici, c'est un « 0 » qui va venir dans la retenue. L'instruction suivante :

BCC BOUCLE

teste la retenue et retourne à BOUCLE tant que la retenue est nulle. C'est notre compteur de bits automatique. On peut voir facilement qu'à la suite du premier ROL, MOT contiendra « 00000001 ». Huit décalages plus tard le « 1 » arrivera dans le bit de retenue et fera cesser les branchements. C'est une manière ingénieuse de réaliser un compteur de boucle automatique sans avoir à consacrer une instruction à décrémenter un registre index. Cette technique est utilisée pour raccourcir le programme et accroître ses performances.

Quand le BCC ne conduit plus à un branchement, c'est que 8 bits ont fini par être assemblés dans la case MOT. Cette valeur doit être

sauvegardée en mémoire. C'est accompli par les instructions qui suivent (flèche 5 de la Figure 6-6).

```
LDA   MOT
PHA
```

Ici, nous sauvons MOT (8 bits) dans la pile. La sauvegarde dans la pile n'est possible que s'il y a assez de place dans la pile. Pourvu que cette condition soit réalisée, c'est la manière la plus rapide de sauvegarder un mot en mémoire. Le pointeur de pile est mis à jour automatiquement. Si nous n'utilisions pas la pile, nous devrions faire appel à une instruction supplémentaire pour mettre à jour un pointeur vers la mémoire. Nous pourrions de façon équivalente utiliser l'adressage indexé, mais cela aussi imposerait d'incrémenter ou de décrémenter l'index, ce qui consommerait du temps.

Lorsque le premier MOT de données a été sauvé, il n'y a plus aucune garantie que le premier bit de données à venir soit un « 1 » il peut être n'importe quoi. Par suite nous devons réinitialiser le contenu de MOT à « 00000001 » pour pouvoir continuer à l'utiliser comme compteur de bits. Ceci est effectué par les deux instructions suivantes :

```
LDA #$01
STA   MOT
```

Finalement, nous décrémentons le compte de mots, puisque nous venons d'assembler un mot, et nous testons si nous avons atteint la fin du transfert. Ceci est accompli par les deux instructions qui suivent :

```
DEC   COMPTE
BNE   BOUCLE
```

Le programme ci-dessous a été conçu pour être rapide, afin de pouvoir saisir un flot de bits de données rapide.

Lorsque le programme se termine, il est, naturellement, recommandé de lire dans la pile les mots qui y ont été sauvegardés et de les transférer ailleurs dans la mémoire. Nous avons déjà appris à effectuer un tel transfert de bloc au Chapitre 2.

Exercice 6.4 : *Calculez la vitesse maximum à laquelle ce programme sera capable de lire des bits en série. Pour calculer cette vitesse, supposez que les adresses MOT et COMPTE sont en page 0. Supposez aussi que tout le programme réside dans une même page. Regardez le nombre de cycles demandés par chaque instruction dans la table de la fin du livre, et calculez le temps écoulé lors de l'exécution du programme. Pour calculer le temps pris par une boucle, multipliez simplement la durée totale de cette boucle (exprimée en microsecondes) par le nombre de fois qu'elle sera exécutée. En outre, pour*

calculer la vitesse maximum supposez qu'un bit de donnée sera prêt à chaque fois qu'on teste le registre d'entrée.

Ce programme est plus difficile à comprendre que les précédents. Examinons le encore (cf. Figure 6-6), avec plus de détails, et en examinant les compromis. Un bit de donnée arrive de temps en temps en position 0 de « ENTREE ». Il peut par exemple, y avoir trois « 1 » successifs. Nous devons, par suite, distinguer les bits successifs qui arrivent. C'est la fonction du signal d'« horloge ».

Le signal d'horloge (ou d'état) nous indique que le bit à l'entrée est maintenant valide.

Avant de lire un bit, nous allons donc d'abord tester le bit d'état. Si l'état est « 0 » nous devons attendre. S'il est « 1 », alors le bit de donnée est valide. Nous supposons ici que le signal d'état est connecté au bit 7 du registre ENTREE.

Exercice 6.5 : *Pouvez-vous expliquer pourquoi on utilise le bit 7 pour l'horloge et le bit 0 pour la donnée ?*

Une fois que nous avons recueilli un bit de donnée, nous voulons le préserver dans un emplacement sûr, puis le décaler à gauche afin de pouvoir saisir le bit suivant.

Malheureusement, dans ce programme on utilise l'accumulateur pour lire et tester à la fois l'horloge et la donnée. Si nous accumulions les données dans l'accumulateur, le bit 7 serait effacé par le bit d'état.

Exercice 6.6 : *Pouvez-vous suggérer une manière de tester l'état sans effacer le contenu de l'accumulateur (à l'aide d'une instruction spéciale) ? Si cela peut être fait, pouvons-nous utiliser l'accumulateur pour accumuler les bits successifs qui arrivent ?*

Exercice 6.7 : *Réécrivez le programme en utilisant l'accumulateur pour stocker les bits qui arrivent. Comparez le au précédent en termes de vitesse et de nombre d'instructions.*

Regardons encore deux autres variantes possibles.

Nous avons supposé que, dans notre exemple, le tout premier bit à venir aurait une signification spéciale, et serait garanti d'être à « 1 ». Mais dans le cas général, il peut avoir n'importe quelle valeur.

Exercice 6.8 : *Modifiez le programme ci-dessus, en supposant que le tout premier bit fait partie des données (et n'a donc pas à être écarté), et peut valoir soit « 0 » soit « 1 » (suggestion : notre « compteur de bits » devrait encore fonctionner correctement, si on l'initialise à la bonne valeur).*

Enfin nous avions mis le MOT assemblé dans la pile pour gagner du temps. Nous pourrions naturellement le ranger dans une zone mémoire spécifiée.

Exercice 6.9 : *Modifiez le programme ci-dessus pour ranger le MOT assemblé dans la zone mémoire commençant à BASE.*

Exercice 6.10 : *Modifiez le programme ci-dessus de façon que le transfert s'arrête lorsque le caractère « S » est détecté dans le flot d'entrée.*

Alternative hardware

Comme d'habitude pour la plupart des algorithmes d'entrée-sortie, il est possible de réaliser la procédure ci-dessus par hardware.

Le boîtier qui le fait s'appelle un UART ou un ACIA (en nomenclature Motorola). Il accumule les bits automatiquement. Mais lorsqu'on souhaite réduire le nombre de composants, on utilise plutôt un programme ou une de ses variantes.

Exercice 6-11 : *Modifier le programme en supposant que la donnée est présentée au bit 0 de l'emplacement ENTREE tandis que l'information d'état est disponible sur le bit 0 de l'adresse ENTREE + 1.*

RECAPITULATION DES ENTREES-SORTIES DE BASE

Nous avons maintenant appris à effectuer les opérations d'entrées-sorties élémentaires ainsi qu'à gérer un flot de données parallèles ou sérielles. Nous sommes maintenant prêts à communiquer avec des périphériques réels.

COMMUNICATION AVEC LES PERIPHERIQUES

Pour échanger des données avec les périphériques, nous devons d'abord nous assurer si les données sont disponibles, lorsque nous voulons lire ou si le périphérique est prêt à accepter des données lorsque nous voulons en envoyer. On peut utiliser deux procédures : la poignée de main et les interruptions. Etudions d'abord la poignée de main.

Synchronisation des échanges (« poignée de main »)

La poignée de main (« handshaking ») est généralement utilisée pour communiquer entre deux appareils asynchrones c'est-à-dire entre deux appareils qui ne sont pas synchronisés automatiquement. Par exemple, si nous voulons envoyer un mot à une imprimante

parallèle, nous devons d'abord nous assurer que le tampon d'entrée de l'imprimante est disponible. Nous allons donc demander à l'imprimante : « êtes-vous prête » ? L'imprimante répondra « oui » ou « non ». Si elle n'est pas prête, nous attendrons. Si elle est prête, nous enverrons les données (voir Figure 6-8).

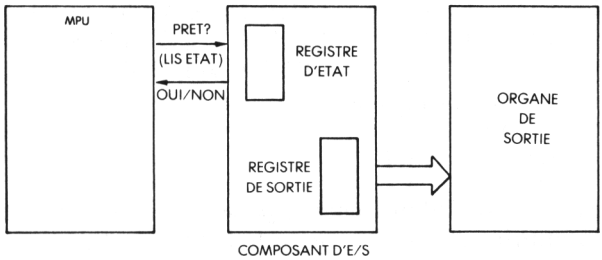


Fig. 6-8 : Poignée de main (sortie)

Inversement, avant de lire des données depuis une unité d'entrée, nous vérifierons si la donnée est valide. Nous demanderons : « la donnée est-elle valide ? ». Et le périphérique nous dira « oui » ou « non ». La réponse peut être fournie par des bits d'état ou par d'autres moyens (voir Figure 6-9).

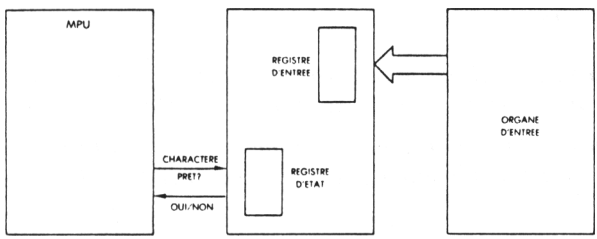


Fig. 6-9 : Poignée de main (Entrée)

En bref, chaque fois que vous voulez échanger des informations avec quelqu'un d'indépendant qui risque d'être en train de faire autre chose à ce moment, vous devez vous assurer qu'il est prêt à communiquer avec vous. La règle de courtoisie habituelle est de lui serrer la main. L'échange de données peut alors suivre. C'est la procédure normalement utilisée pour les communications avec les périphériques.

Illustrons maintenant cette procédure avec un exemple simple :

Envoi d'un caractère à une imprimante

On supposera que le caractère est contenu dans la case-mémoire CAR. Le programme qui l'imprime apparaît ci-dessous.

```
IMPCAR  LDX  CAR      LIRE LE CARACTERE
.ATTEND LDA  ETAT     LE BIT 7 CONTIENT
                      L'INDICATEUR « PRET »
        BPL  ATTEND
        TXA
        STA  IMPRD
```

Le registre X est d'abord chargé à partir de la mémoire avec le caractère à imprimer. Ensuite, on teste le bit d'état de l'imprimante pour déterminer si elle est prête à accepter le caractère. Tant qu'elle n'est pas prête à imprimer, on retourne à l'adresse ATTEND et on boucle.

Quand l'imprimante indique qu'elle est prête à imprimer en positionnant son bit « prêt » (ici par convention, le bit 7 de l'adresse ETAT), nous pouvons envoyer le caractère. Nous le transférons du registre X dans A :

```
TXA
```

et nous l'envoyons au registre donnée de l'imprimante, appelé ici IMPRD :

```
STA IMPRD
```

Exercice 6.12 : *Modifier le programme ci-dessus pour imprimer une chaîne de n caractères où on supposera n inférieur à 255.*

Exercice 6.13 : *Modifier le programme ci-dessus pour imprimer une chaîne de caractères jusqu'à ce qu'on rencontre le code « retour-chariot ».*

Compliquons maintenant la procédure de sortie en demandant une conversion de code et en commandant plusieurs périphériques en même temps.

Sortie sur un afficheur LED sept-segments

Un afficheur à diodes électroluminescentes (LED) 7 segments traditionnel peut afficher les chiffres « 0 » à « 9 », ou même « 0 » à « F » en hexadécimal, en éclairant des combinaisons de ses 7 segments. Une LED 7 segments est montrée sur la Figure 6-10. Les caractères qui peuvent être formés par cette LED apparaissent Figure 6-11.

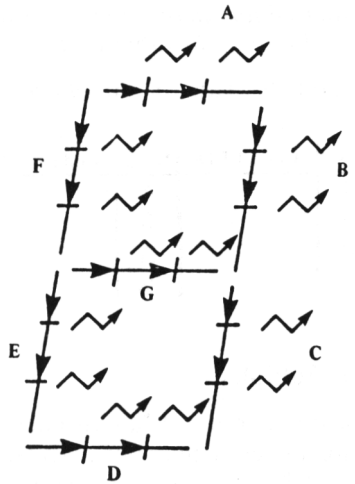


Fig. 6-10 : LED sept-segments

Les segments d'une LED sont désignés de « a » à « g » comme le montre la Figure 6-10. Par exemple, on affichera « 0 » en éclairant les segments « abcdef ». Supposons maintenant que le bit 0 d'un port de sortie est connecté au segment « a », que le bit 1 est connecté au segment « b », et ainsi de suite. Le bit 7 n'est pas utilisé. Le code binaire permettant d'éclairer « fedcba » (pour afficher « 0 ») est donc « 0111111 ». En hexadécimal, c'est « 3F ». Faites l'exercice suivant :

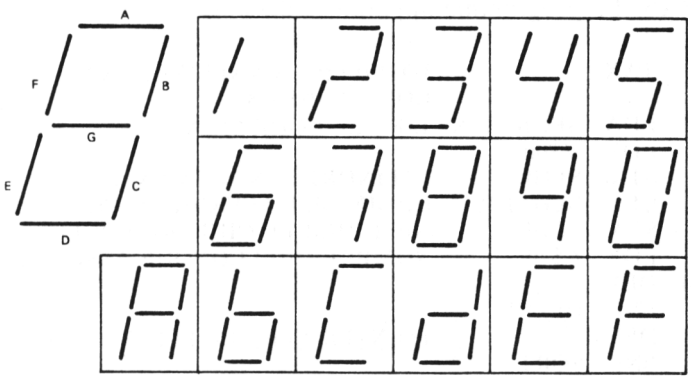


Fig. 6-11 : Caractères formés avec une led 7 segments

Exercice 6.14 : *Calculer l'équivalent 7 segments des chiffres hexadécimaux « O » à « F ». Remplir la table ci-dessous :*

Affichons maintenant des valeurs sur *plusieurs* LED.

	Code LED	Hexa	Code LED	Hexa	Code LED	Hexa	Code LED
0	3F	4		8		C	
1		5		9		D	
2		6		A		E	
3		7		B		F	

Commande de LED multiples

Les LED n'ont pas de mémoire. Elles affichent les données seulement lorsque leurs lignes-segments sont actives. Pour maintenir le coût d'un afficheur à LED assez bas, le microprocesseur affiche l'information sur *chaque LED à son tour.* La rotation entre les LED doit être assez rapide pour qu'il n'y ait pas de scintillement. Ceci implique que le temps passé pour aller d'une LED à la suivante soit inférieur à 100 millisecondes. Réalisons un programme qui accomplisse ceci. Nous utiliserons le registre Y pour pointer vers la LED sur laquelle nous voulons afficher un chiffre. L'accumulateur est supposé contenir la valeur hexadécimale à afficher sur la LED. Notre premier problème est de convertir la valeur hexadécimale dans sa représentation 7 segments. Nous avons construit la table d'équivalence dans la section précédente. Comme nous accédons à une table, nous utiliserons l'adressage indexé, avec, comme déplacement, la valeur hexadécimale. Cela signifie que le code 7 segments pour le chiffre hexadécimal n° 3 sera obtenu en prenant le 3e élément de la table après la base. L'adresse de la base sera appelée SEGBAS. Le programme apparaît ci-dessous :

```
        TAX                 .UTILISER LA VALEUR HEXA
                            COMME INDEX
LED     LDA  SEGBAS ,X      LIRE LE CODE EN A
        LDX  #$00
        STX  SEGD           ETEINDRE LES SEGMENTS
        STY  SEGADR         CHOIX DE L'AFFICHEUR CONCERNE
        STA  SEGD           AFFICHER LE CHIFFRE VOULU
        LDX  #$70           NOMBRE QUELCONQUE ASSEZ GRAND
DELAI   DEX
        BNE  DELAI
        DEY                 POINTER VERS LA LED SUIVANTE
        BNE  SORT
        LDY  NBLED
SORT    RTS
```

Le programme suppose que le registre Y contienne le numéro du prochain afficheur à éclairer et que le registre X contienne le chiffre à afficher.

Le programme commence par prendre le code 7 segments correspondant à la valeur contenue dans l'accumulateur (deux premières instructions). Les deux instructions suivantes chargent « 00 » comme valeur des segments à afficher, c'est-à-dire les éteignent tous. L'instruction suivante choisit l'afficheur approprié : STY SEGADR, et la suivante lui envoie le code 7 segments à afficher : STA SEGD.

Un délai est alors marqué par une boucle de trois instructions avant de passer à l'afficheur suivant. Enfin le pointeur vers l'afficheur est décrémenté (il pourrait être incrémenté).

Si le pointeur de LED décrémente à « 0 », il doit être rechargé avec le numéro de LED le plus élevé. C'est ce que font les deux instructions qui suivent. On suppose ici qu'il s'agit d'un sous-programme, et la dernière instruction est un RTS : « retour de sous-programme ».

Exercice 6.14 : *Supposant que le programme ci-dessus forme un sous-programme, vous remarquerez qu'il utilise les registres internes X et Y et les modifie. Supposant que le sous-programme peut librement utiliser la zone-mémoire d'adresses T1, T2, T3, T4 et T5, pouvez-vous ajouter des instructions au début et à la fin de ce programme qui garantissent que lorsqu'on revient du sous-programme, le contenu des registres X et Y sera le même que lorsqu'on est entré ?*

Exercice 6.15 : *Même exercice que ci-dessus mais en supposant que le sous-programme n'a pas la disposition de la zone-mémoire T1, ... etc. (Suggestion : rappelez-vous qu'il y a dans tout ordinateur un mécanisme incorporé qui permet de préserver des informations dans l'ordre chronologique).*

Nous avons maintenant résolu les problèmes d'entrées-sorties usuels. Considérons le cas d'un périphérique réel : le Télétype.

Entrées-sorties sur Télétype

Le Télétype est un périphérique sériel. Il envoie et reçoit des mots d'information sous forme sérielle. Chaque caractère est codé suivant le format ASCII 8 bits (la table ASCII apparaît à la fin de ce livre).

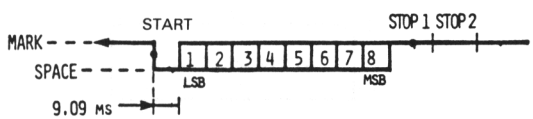


Fig. 6-12 : Format d'un mot sur Télétype

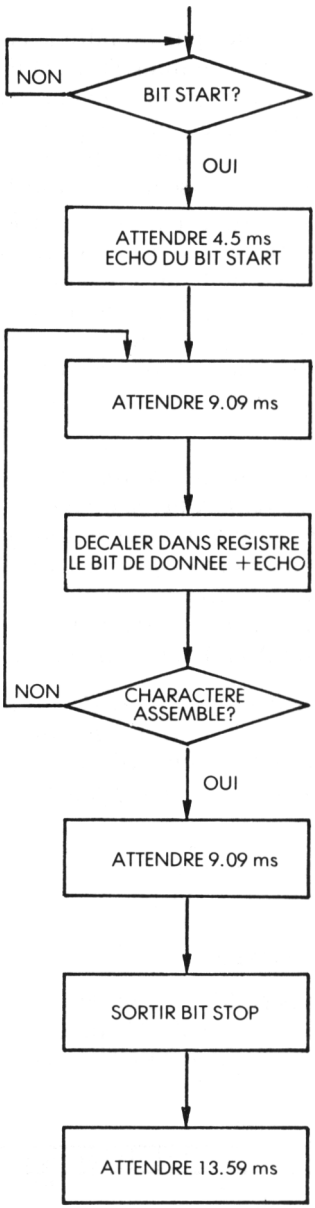


Fig. 6-13 : Entrée sur TTY avec écho

En outre, chaque caractère est précédé par un bit de « départ » (« START »), et terminé par deux bits d'« arrêt » (« STOP »). Dans l'interface, dite boucle de courant 20 mA qui est la plus utilisée, l'état de la ligne est normalement à « 1 ». Cela sert à indiquer au processeur que la ligne n'est pas coupée. Le départ est marqué par une transition de « 1 » vers « 0 ». Il indique au récepteur que les bits de données suivent. Le Télétype standard fonctionne à 10 caractères par seconde. Nous venons d'établir que chaque caractère requiert 11 bits. Cela signifie que le Télétype transmet à 110 bits par seconde. On parle de 110 baud. Nous allons écrire un programme qui sérialise les bits vers le Télétype à la vitesse voulue.

110 bits par seconde implique que les bits sont séparés par 9,09 millisecondes. Telle sera la durée de la boucle de délai à réaliser entre les bits successifs. Le format d'un mot sur Télétype apparaît Figure 6-12. L'ordinogramme du programme d'entrée apparaît Figure 6-13.

Le programme est le suivant :

```
TTYIN    LDA  ETAT
         BPL  TTYIN     SCRUTATION HABITUELLE DE L'ETAT
         JSR  DELAI     ATTENDRE 9,09 MS
         LDA  TTY       BIT DE DEPART
         STA  TTY       LE RENVOYER EN ECHO
         JSR  DELAI
         LDX  #$ 08     COMPTEUR DE BITS
SUIVANT  LDA  TTY
         STA  TTY       ECHO
         LSR  A         SAUVER LE BIT DANS LA RETENUE
         ROL  CAR       SAUVEGARDE DU BIT DANS CAR
         JSR  DELAI
         DEX            BIT SUIVANT
         BNE  SUIVANT
         LDA  TTY       BIT D'ARRET
         STA  TTY       ECHO
         JSR  DELAI
         RTS
```

Fig. 6-14 : Entrée Télétype

Notez que ce programme est légèrement différent de l'ordinogramme de la Fig. 6-13. Le programme doit être examiné avec attention. La logique en est très simple. Le fait nouveau est que, chaque fois qu'un bit est lu depuis le Télétype (à l'adresse TTY), il est renvoyé en écho sur le Télétype. C'est une particularité standard du Télétype. Chaque fois qu'un utilisateur appuie sur une touche, l'information est transmise au processeur, puis renvoyée au mécanisme d'impression du Télétype. Lorsqu'un caractère s'imprime correcte-

ment sur le papier cela permet de vérifier que les lignes de transmission fonctionnent bien et que le processeur marche.

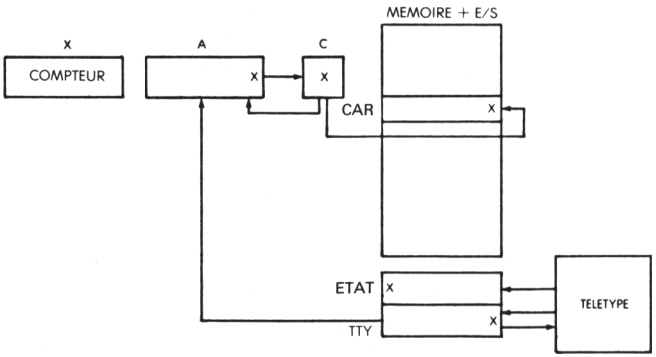


Fig. 6-15 : Entrée Télétype

Les deux premières instructions forment une boucle d'attente. Le programme attend que le bit d'état devienne « vrai » avant de commencer à lire les bits. Comme d'habitude, le bit d'état est supposé être à la position 7, car cette position peut être testée en une seule instruction avec un BPL (« branchement si plus », qui teste le bit de signe).

JSR est un appel de sous-programme. Nous utilisons un sous-programme DELAI pour réaliser le délai de 9,09 ms. Notez que DELAI peut être une boucle programmée ou faire appel à un temporisateur hardware, s'il y en a un dans notre système.

Le premier bit qui arrive est le bit de départ. Il doit être renvoyé en écho vers le Télétype, mais, à part cela, il est ignoré. Ceci est accompli par les instructions 4 et 5.

Là encore, nous attendons le prochain bit. Mais, cette fois, il s'agit d'un vrai bit de donnée, et nous devons le sauvegarder. Comme toutes les instructions de décalage envoient un bit dans l'indicateur de retenue, il nous faut deux instructions pour préserver notre bit de donnée (le « X » de la Figure 6-15) : une pour l'envoyer dans la retenue C (« LSR A »), et une pour le préserver dans la case-mémoire CAR (ROL).

Finalement nous faisons l'écho du bit de donnée : STA TTY et nous attendons le suivant : JSR DELAI

jusqu'à avoir accumulé 8 bits : DEX.

Quand la décrémentation arrive à « 0 », les 8 bits sont dans CAR. Nous n'avons plus qu'à renvoyer les bits d'arrêt et c'est terminé.

Exercice 6.16 : *Ecrire le sous-programme DELAI qui réalise le délai de 9,09 ms.*

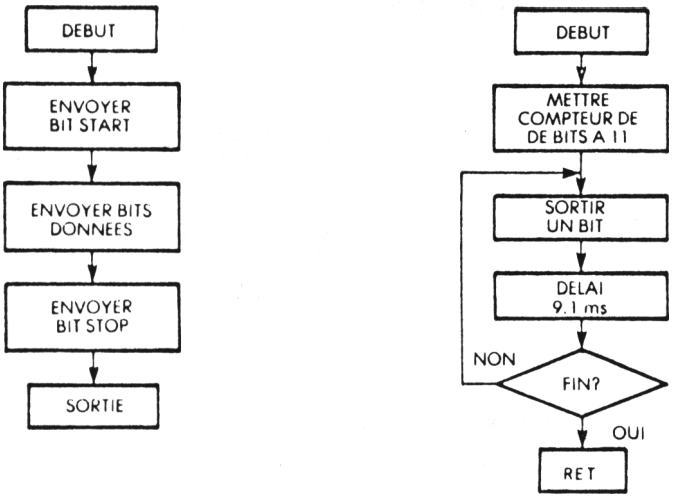


Fig. 6-16 : Sortie sur Télétype

Exercice 6.17 : *A l'aide de l'exemple du programme développé ci-dessus, écrire un programme IMPRC qui imprime sur Télétype le contenu de l'emplacement-mémoire CAR.*

Exercice 6.18 : *Modifier le programme pour qu'il attende un bit de DEPART et non un bit d'état.*

Impression d'une chaîne de caractères

Nous supposerons que la routine IMPRC (cf. exercice 6.17) se charge d'imprimer un caractère sur notre imprimante ou de l'afficher sur n'importe quel organe de sortie. Ici, nous imprimerons le contenu des cases-mémoire de DEPART + N à DEPART.

Nous utiliserons naturellement l'adressage indexé et le programme est évident :

```
IMCHAIN   LDX   #N              NOMBRE DE MOTS
SUIVANT   LDA   DEPART,X
          JSR   IMPRC
          DEX
          BPL   SUIVANT
```

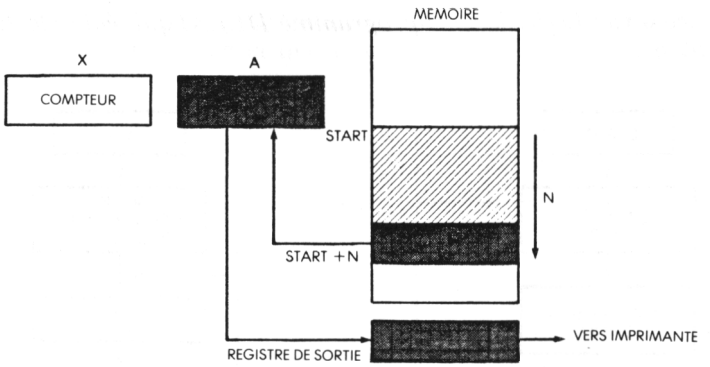


Fig. 6-17 : Impression d'un bloc-mémoire

RECAPITULATION DES PERIPHERIQUES

Nous avons maintenant décrit les techniques de programmation de base utilisées pour communiquer avec des périphériques type. Outre les transferts de données proprement dits, il est nécessaire de positionner un ou plusieurs registres de commande à l'intérieur de chaque boîtier d'entrée-sortie afin de préparer correctement les vitesses de transfert, les mécanismes d'interruption et les différentes autres options. Il faut consulter le manuel de chaque périphérique (pour plus de détail sur les algorithmes spécifiques d'échange d'informations avec tous les périphériques usuels, le lecteur peut se référer à notre livre C5 : techniques d'interface aux microprocesseurs).

Nous avons appris à gérer des périphériques isolés. Mais, dans un système réel, tous les périphériques sont connectés aux bus et peuvent demander qu'on s'occupe d'eux simultanément. Comment allons-nous répartir le temps du processeur ?

ORGANISATION DES ENTREES/SORTIES

Comme les requêtes d'entrée-sortie peuvent arriver simultanément, il faut dans tout système implanter un mécanisme d'organisation capable de déterminer dans quel ordre les demandes seront servies. Trois techniques d'entrée-sortie de base sont utilisées et elles peuvent être combinées. Ce sont : la scrutation, les interruptions, le DMA. On discutera ici de la scrutation et des interruptions. Le DMA est une technique de hardware pur ; il ne sera donc pas décrit ici. (Il est traité dans les livres de référence C4 et C5).

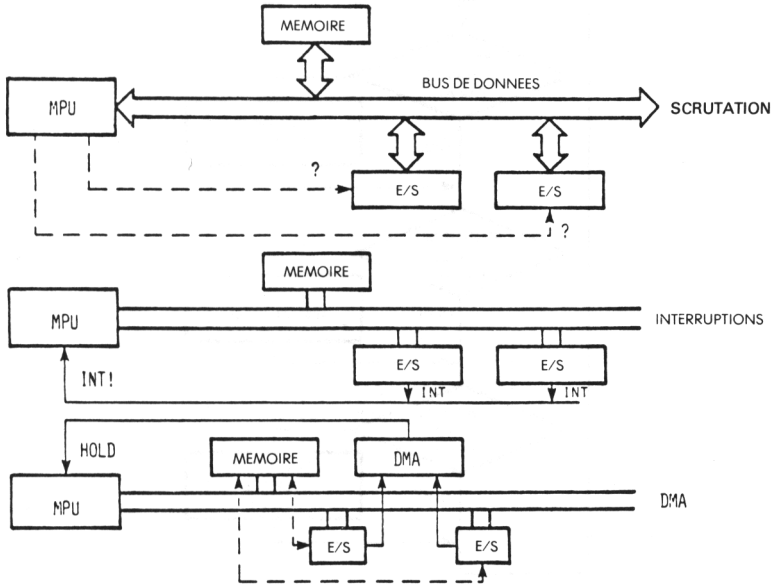


Fig. 6-18 : Les trois méthodes de gestion des entrées-sorties

Scrutation

Conceptuellement, la scrutation (ou sondage — en anglais : « polling ») est la méthode la plus simple pour gérer des périphériques multiples. Dans cette stratégie, le processeur interroge tour à tour les unités connectées au bus. Si une unité demande du service, celui-ci est accordé. Si l'unité n'en demande pas, on examine le périphérique suivant. La scrutation n'est pas utilisée simplement pour les périphériques, mais pour toute boucle de service.

Par exemple, si le système comporte un Télétype, une unité de bande et un écran, la routine de scrutation va interroger, d'abord le Télétype : « avez-vous un caractère à transmettre » ? Elle va aussi interroger la *routine de sortie* sur Télétype en lui demandant : « avez-vous un caractère à envoyer » ?

Ensuite, supposant que les réponses sont jusqu'ici négatives, elle va interroger les routines du dérouleur de bandes, puis de l'écran. Dans le cas où il y a un seul périphérique connecté au système la scrutation sera également utilisée pour déterminer s'il a besoin d'être servi. Par exemple, les ordinogrammes de lecture sur un lecteur de ruban perforé et d'écriture sur imprimante apparaissent sur les Figures 6-20 et 6-21.

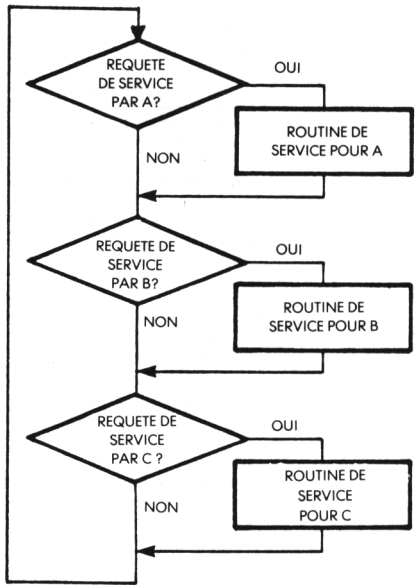


Fig. 6-19 : Ordinogramme de la boucle de scrutation

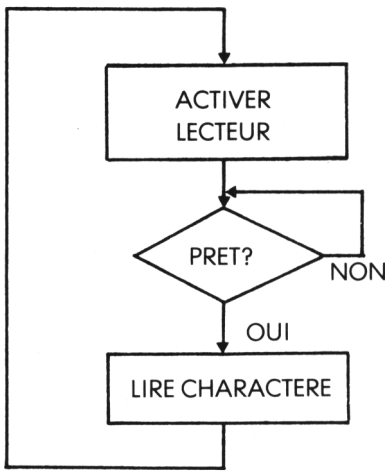


Fig. 6-20 : Lecture sur un lecteur de ruban perforé

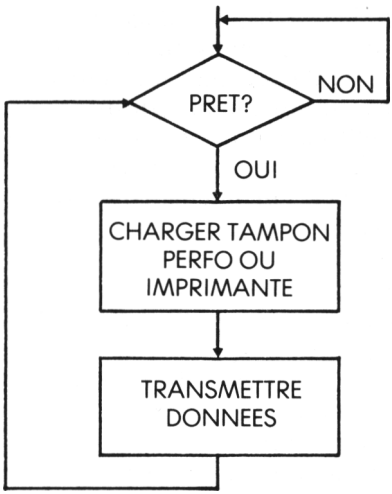


Fig. 6-21 : Ecriture sur perforateur ou imprimante

Exemple : boucle de scrutation pour les périphériques 1, 2, 3, 4 (cf. Figure 6-18).

```
SCRUT4   LDA   ETAT1    PERIPHERIQUE 1 ?
         BMI   UN       LE BIT DE DEMANDE
                        DE SERVICE EST LE BIT 7
         LDA   ETAT2    PERIPHERIQUE 2 ?
         BMI   DEUX
         LDA   ETAT3    PERIPHERIQUE 3 ?
         BMI   TROIS
         LDA   ETAT4    PERIPHERIQUE 4 ?
         BMI   QUATRE
         JMP   SCRUT4   REBOUCLER
```

Le bit 7 du registre d'état de chaque périphérique est à 1 lorsqu'il a besoin d'être servi. Quand une requête est décelée, ce programme se branche vers la routine du périphérique à l'adresse UN pour le périphérique 1, DEUX pour le périphérique 2... etc.

Les avantages de la scrutation sont évidents : elle est simple, ne demande aucun hardware, et conserve la synchronisation de toutes les entrées-sorties avec le fonctionnement du programme. Son inconvénient est tout aussi évident : la plupart du temps du processeur est gaspillée à interroger des périphériques qui n'ont pas besoin d'être servis. En outre le processeur risque de s'occuper trop tard d'un périphérique à force de perdre du temps.

Il est donc désirable de disposer d'un autre mécanisme qui garantisse de pouvoir utiliser le temps du processeur à effectuer des calculs utiles plutôt que d'interroger tout le temps des périphériques qui n'en ont pas besoin. Néanmoins, insistons sur le fait que la scrutation est universellement utilisée dès que le microprocesseur n'a rien de mieux à faire, étant donné qu'elle garde l'organisation globale simple. Examinons maintenant l'alternative essentielle de la scrutation : les interruptions.

Interruptions

Le concept d'interruption est illustré sur la Figure 6-18. On dispose d'une ligne hardware spéciale, la ligne interruption, qui est connectée à une broche spéciale du microprocesseur. De multiples organes d'entrée-sortie peuvent être reliés à cette ligne d'interruption. Quand l'un d'entre eux a besoin d'être servi, il envoie un niveau ou une impulsion sur cette ligne. Le signal d'interruption est la requête de service émanant d'un organe d'entrée-sortie vers le processeur. Examinons la réponse du processeur à cette interruption.

Dans tous les cas, le processeur termine l'instruction qu'il était en train d'exécuter, sinon cela créerait le chaos à l'intérieur du microprocesseur. Ensuite, le processeur doit se brancher à une routine de traitement de l'interruption qui va s'occuper de l'interruption. Le branchement à un tel sous-programme implique que le contenu du compteur ordinal soit sauvegardé dans la pile. *Une interruption doit donc entraîner la sauvegarde automatique du compteur ordinal dans la pile.* En outre, le registre d'état (P) doit, lui aussi, être sauvé automatiquement, puisque son contenu sera altéré par toute instruction subséquente. Enfin si la routine de service de l'interruption doit modifier des registres internes, ceux-ci doivent eux aussi être préservés dans la pile.

Après que tous ces registres aient été sauvegardés, on peut se brancher à l'adresse appropriée de la routine d'interruption. A la fin de cette routine, tous les registres doivent être restaurés et on doit exécuter une instruction spéciale de retour d'interruption afin de reprendre l'exécution du programme principal. Examinons plus en détail les deux lignes d'interruption du 6502.

Interruptions du 6502

Le 6502 possède deux lignes d'interruption, IRQ et NMI. IRQ est la ligne d'interruption normale, tandis que NMI est une interruption plus prioritaire et non masquable. Examinons leur fonctionnement. IRQ est une interruption activée par niveau. L'état de la ligne IRQ sera testé ou ignoré par le microprocesseur selon la valeur de l'indicateur interne I (indicateur de masque d'interruption). Nous suppo-

serons d'abord que les interruptions sont autorisées. A chaque fois que IRQ sera activée, l'interruption sera détectée par le microprocesseur. Dès que l'interruption est acceptée (après achèvement de l'instruction en cours), l'indicateur I est automatiquement mis à 1. Cela empêchera le microprocesseur d'être interrompu à nouveau au moment où il manipule les registres internes.

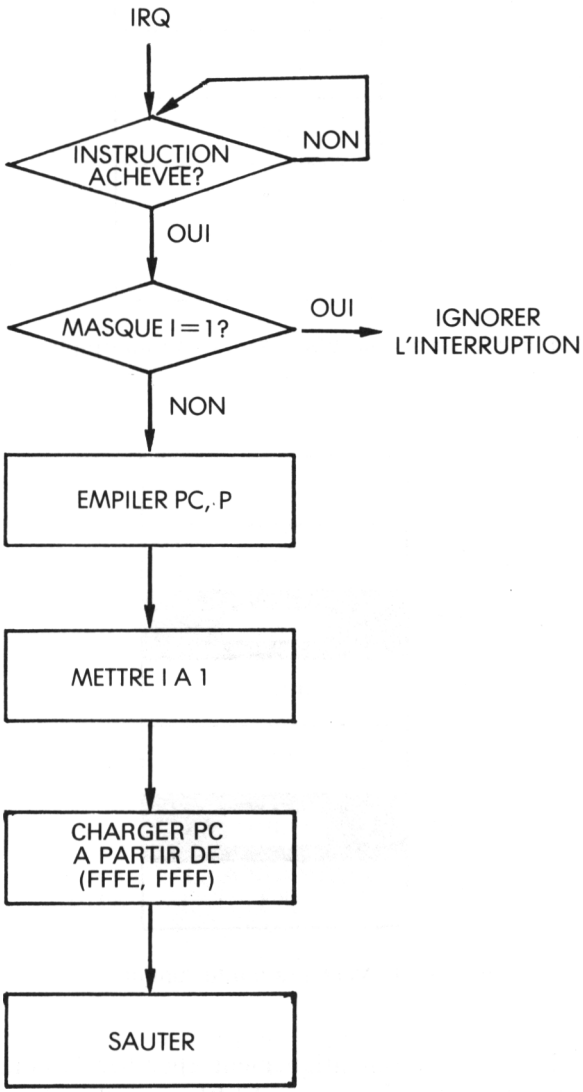


Fig. 6-22 : Traitement des interruptions

Ensuite, le 6502 préserve automatiquement le contenu de PC (compteur ordinal) et celui de P (registre d'état) dans la pile. L'aspect de la pile après une interruption est illustré par la Figure 6-23.

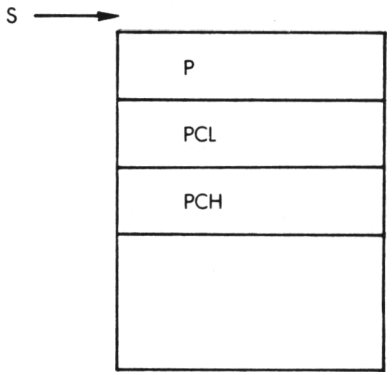


Fig. 6-23 : La pile du 6502 après une interruption

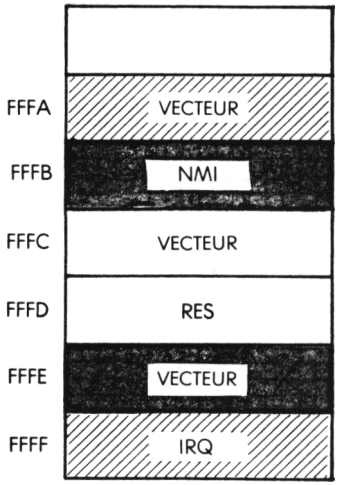


Fig. 6-24 : Vecteurs d'interruption

Ensuite, le 6502 va automatiquement chercher le contenu des adresses-mémoire « FFFE » et « FFFF ». Cet emplacement-mémoire de 16 bits contient le *vecteur d'interruption*. Le 6502

recherche le contenu de cette adresse, puis se branche au vecteur de 16 bits spécifié. Il est de la responsabilité de l'utilisateur de déposer cette adresse de vectorisation en « FFFE », « FFFF ». Cependant, plusieurs périphériques peuvent être connectés à la ligne IRQ. Dans ce cas nous nous branchons à une seule routine de service de l'interruption. Comment allons-nous faire la distinction entre les différents périphériques ? Ceci sera étudié dans la section suivante.

L'interruption NMI est pour l'essentiel identique à IRQ, sauf qu'elle ne peut être masquée par le bit I. C'est une interruption prioritaire utilisée généralement pour les pannes de courant. Le fonctionnement à part cela est identique, sauf que le processeur se branche automatiquement sur le contenu de « FFFA », « FFFB ». Ceci est illustré Figure 6-24.

Le retour d'interruption est effectué par l'instruction RTI. Cette instruction récupère dans le microprocesseur les trois mots du haut de la pile qui contiennent P et PC (le compteur ordinal 16 bits). Le programme qui avait été interrompu peut alors reprendre.

L'état interne de la machine est exactement identique à ce qu'il était lorsque l'interruption est arrivée. L'effet a été d'introduire un retard dans l'exécution du programme. Il est de la responsabilité du programmeur, avant de revenir d'une interruption, d'acquitter l'interruption qui vient d'être traitée, et de restaurer l'indicateur d'inhibition des interruptions. En outre, si la routine d'interruption modifie le contenu de certains registres, comme X ou Y, il appartient spécifiquement au programmeur de sauver ces registres dans la pile avant d'exécuter le traitement proprement dit de l'interruption. Sinon, le contenu de ces registres serait perdu et lorsque le programme interrompu reprendrait son exécution, il serait faux.

Si on suppose que la routine d'interruption utilise les registres A, X et Y, il faut 5 instructions dans la routine de gestion d'interruption pour les préserver. Ce sont :

```
SAVAXY   PHA    EMPILER A
         TXA    TRANSFERER X EN A
         PHA    EMPILER
         TYA    TRANSFERER Y EN A
         PHA    EMPILER
```

Malheureusement, le 6502 ne peut empiler directement que le contenu de A ou de P. Il en résulte que la sauvegarde de X et de Y prend du temps : elle exige 4 instructions. Ceci est illustré Figure 6-25.

A la fin de la routine d'interruption, il faut restaurer ces registres et la routine d'interruption doit se terminer par la suite de 6 instructions :

```
PLA     DEPILER Y
TAY     RESTAURER Y
PLA     DEPILER X
TAX     RESTAURER X
PLA     RESTAURER A
RTI     RETOUR D'INTERRUPTION
```

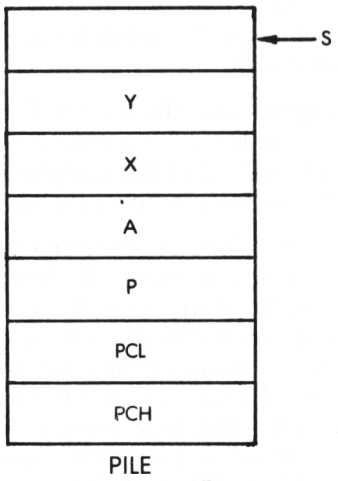


Fig. 6-25 : Sauvegarde de tous les registres

Exercice 6.19 : *A l'aide de la table de l'appendice qui indique le nombre de cycles par instruction, calculer combien de temps sera perdu à sauvegarder puis à restaurer les registres A, X et Y.*

Pour un graphique de comparaison entre le processus de scrutation et celui de l'interruption, on se reportera à la Figure 6-18 où la scrutation est illustrée au-dessus et l'interruption en-dessous. On peut voir que, dans la technique de scrutation, le programme perd beaucoup de temps à attendre. Avec les interruptions, le programme est interrompu, l'interruption est traitée, puis le programme reprend. Néanmoins, l'inconvénient évident de l'interruption est d'introduire plusieurs instructions supplémentaires au début et à la fin ce qui entraîne un délai avant que la première instruction de la routine de gestion du périphérique puisse être exécutée. C'est un temps perdu supplémentaire.

Ayant clarifié le fonctionnement des deux lignes d'interruptions, considérons maintenant les deux problèmes importants qui demeurent :

1. Comment résoudre le problème de plusieurs périphériques déclenchant une interruption en même temps ?

2. Comment résoudre le problème de l'arrivée d'une interruption alors qu'on est en train de traiter une autre interruption ?

Périphériques multiples connectés à une même ligne d'interruption

A chaque fois qu'une interruption arrive, le processeur se branche à une adresse contenue en FFFE-FFFF (pour une IRQ) ou en FFFA-FFFB (pour un NMI). Avant de pouvoir faire un traitement effectif quelconque, la routine d'interruption doit déterminer quel est le périphérique qui a déclenché l'interruption. Comme d'habitude, on dispose de deux méthodes pour identifier le périphérique : une méthode software et une méthode hardware.

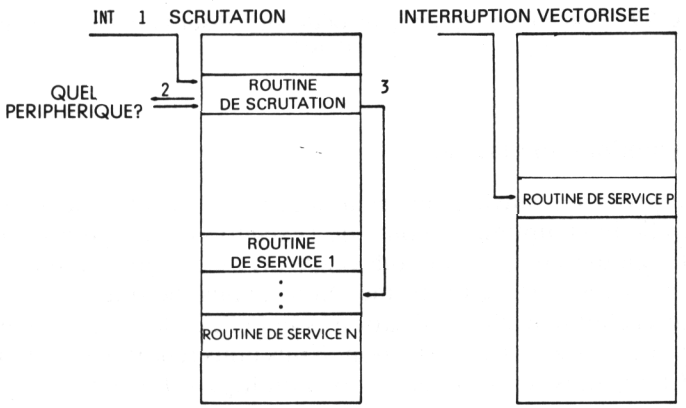


Fig. 6-26 : Interruption vectorisée contre scrutation

La méthode software fait appel à la scrutation. Le microprocesseur interroge tour à tour chaque périphérique en lui demandant : « Est-ce toi qui a déclenché l'interruption » ? Sinon, il interroge le suivant. Ce processus est illustré sur la Figure 6-26. Un programme type est :

```
LDA    ETAT1
BMI    UN
LDA    ETAT2
BMI    DEUX
```

La méthode hardware utilise des composants supplémentaires, mais elle fournit l'adresse du périphérique qui active l'interruption en même temps que la requête d'interruption. Le circuit utilisé maintenant de façon universelle pour cet usage est appelé un « PIC » (contrôleur de priorité d'interruptions). Un tel PIC place automatiquement sur le bus des données l'adresse de branchement exacte voulue pour le périphérique concerné. Quand le 6502 ira en FFFE-FFFF, il va chercher cette adresse de vectorisation. Le concept est illustré sur la Figure 6-26. Dans la plupart des cas, la vitesse de réponse à une interruption n'est pas cruciale et on utilise la scrutation. Si le temps de réponse est de première importance il faut employer la solution hardware.

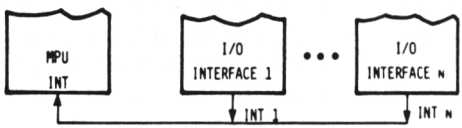


Fig. 6-27 : Plusieurs périphériques peuvent utiliser la même ligne d'interruption

Interruptions simultanées

Le problème suivant qui peut surgir est le déclenchement d'une nouvelle interruption au cours de l'exécution d'une routine d'interruption antérieure. Examinons ce qui arrive et comment on utilise la pile pour résoudre le problème. Nous avons indiqué au Chapitre 2 que ceci était une autre utilisation essentielle de la pile, et le temps est venu d'en faire la démonstration. Nous nous reporterons à la Figure 6-28 pour illustrer les interruptions multiples. Le temps s'écoule de gauche à droite sur la figure. Le contenu de la pile apparaît au bas de l'illustration. En se reportant à gauche, à l'instant T0, le programme est en cours d'exécution. Si on se déplace vers la droite, à l'instant T1 survient l'interruption I1. Nous supposerons que le masque d'interruption était positionné de façon à autoriser I1. Le programme P est suspendu. Cela apparaît au bas de l'illustration. La pile va contenir au moins le compteur ordinal et le registre d'état du programme P plus tout registre éventuel qui pourrait être sauvegardé par la routine d'interruption, ou I1 elle-même.

A l'instant T1, l'interruption I1 démarre et s'exécute jusqu'à T2. A l'instant T2, survient l'interruption I2. Nous supposerons que I2 est considérée comme plus prioritaire que I1. Si elle avait une priorité plus basse, elle aurait été ignorée jusqu'à la fin de I1.

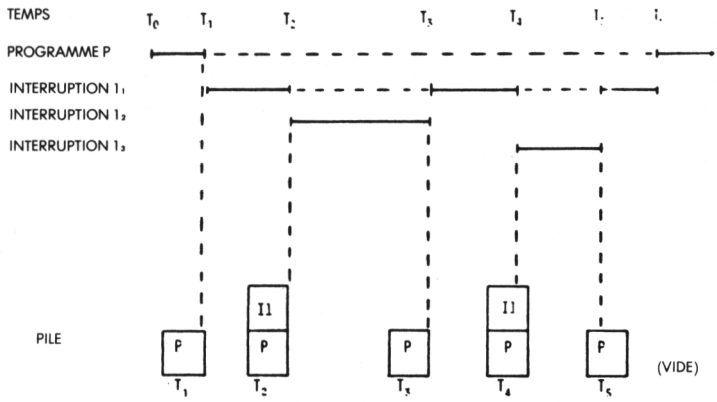


Fig. 6-28 : La pile pendant les interruptions

A l'instant T2, les registres du contexte de I1 sont empilés et cela apparaît au bas de la figure. Là encore, le contenu du compteur ordinal et de P est mis dans la pile. En outre, la routine de traitement de I2 pourrait décider de sauvegarder quelques registres supplémentaires.

I2 va maintenant s'exécuter jusqu'à la fin, à l'instant T3.

Quand I2 se termine, le contenu de la pile est automatiquement renvoyé dans le 6502 et cela apparaît en bas de la Figure 6-28. Ainsi l'interruption I1 reprend son exécution. Malheureusement, à l'instant T4, une interruption plus prioritaire I3 survient de nouveau. Nous pouvons voir en bas de la figure, que, de nouveau, le contexte de I1 est empilé. L'interruption I3 est traitée de T4 à T5 et se termine en T5. A cet instant, le sommet de la pile est dépilé dans le 6502 et l'interruption I1 reprend. Cette fois elle arrive à la fin et se termine en T6. En T6 les registres qui restaient dans la pile sont renvoyés dans le 6502 et le programme P peut reprendre son exécution. Le lecteur vérifiera qu'à cet instant la pile est vide. En fait, le nombre de lignes pointillées qui indique des suspensions de programme, indique en même temps combien de niveaux sont présents à cet instant dans la pile.

Exercice 6.19 : *Si nous supposons qu'à chaque fois qu'une interruption se produit, le compteur ordinal PC, le registre P et l'accumulateur seront sauvegardés, cela occupera un minimum de 4 octets. (En pratique X et Y peuvent être sauvegardés, aussi ce qui demanderait 6 octets). Supposant donc qu'on ne sauvegarde dans la pile que 3 registres, combien de niveaux d'interruptions le 6502 autorise-t-il ? (Rappelez-vous que la pile est limitée à 256 octets en page 1).*

Exercice 6.20 : *Supposant cette fois qu'on sauvegarde 5 registres dans la pile, quel est le nombre maximum d'interruptions simultanées qu'on peut traiter ? Y-a-t-il d'autres facteurs qui contribuent à diminuer encore le nombre d'interruptions simultanées ?*

Il faut néanmoins insister sur le fait que, en pratique, les systèmes à microprocesseur sont normalement connectés à un petit nombre de périphériques utilisant les interruptions. Il est par suite improbable qu'il se produise un grand nombre d'interruptions simultanées dans un tel système.

Nous avons maintenant résolu tous les problèmes généralement associés aux interruptions. En fait, leur utilisation est simple et même le programmeur novice aurait avantage à les utiliser. Complétons notre analyse des ressources du 6502 en introduisant une instruction supplémentaire dont l'effet est de produire une interruption synchrone.

BRK

L'instruction BRK du 6502 est équivalente à une interruption software. Elle peut être insérée dans un programme et entraîne exactement comme dans le cas d'une IRQ, la préservation automatique de PC et P, puis un branchement indirect à FFFE-FFFF. Cette instruction est avantageuse à utiliser pour créer des interruptions programmées lors de la mise au point d'un programme. Il en résultera un point d'arrêt, stoppant le programme à un endroit prévu et se branchant à une routine qui, normalement permettra à l'utilisateur d'analyser le programme. Comme l'effet global du BRK et d'une interruption est le même une fois qu'ils ont eu lieu, il faut fournir un moyen de déterminer si c'était une interruption ou un BRK. Le 6502 positionne un indicateur, B, du registre P (sauvé dans la pile) à « 1 » si c'était un BRK et à « 0 » si c'était une interruption. On peut tester l'état de ce bit à l'aide du programme simple suivant :

```
TEST B  PLA          LIRE LE SOMMET
                     DE LA PILE EN A
        PHA          LE REECRIRE
        AND #$10     MASQUER LE BIT B
        BNE PRBRK    ALLER AU PROGRAMME DE BRK
```

Ce programme de test est normalement inséré à la fin de la séquence de scrutation qui détermine le périphérique qui avait déclenché l'interruption.

Attention : une particularité du BRK est de préserver automatiquement le contenu du compteur ordinal *plus 2.*

Comme le BRK n'occupe qu'un octet, le programmeur peut avoir quelquefois à ajuster le contenu du compteur ordinal dans la pile à l'aide d'une instruction d'incrémentation ou de décrémentation afin de reprendre l'exécution à l'adresse correcte. En particulier, le BRK est utilisé de façon intensive lors des mises au point, en l'écrivant à la place d'une instruction du programme. Si le programme est réassemblé avant l'exécution, il faudra normalement décrémenter de 1 le contenu du compteur ordinal qui aura été sauvegardé.

RECAPITULATION

Nous avons présenté dans ce chapitre l'ensemble des techniques utilisées pour communiquer avec le monde extérieur. Depuis les routines élémentaires d'entrée-sortie jusqu'aux programmes plus complexes pour communiquer avec les périphériques existants, nous avons appris à développer tous les programmes usuels et avons même examiné l'efficacité de programmes de benchmark dans le cas d'un transfert parallèle et dans celui d'une conversion série-parallèle. Enfin nous avons appris à organiser le fonctionnement de périphériques multiples par scrutation ou par interruptions. Naturellement beaucoup d'autres périphériques de toutes sortes pourraient être connectés à un système. Avec l'ensemble des techniques qui ont été présentées jusqu'ici et moyennant la compréhension des périphériques concernés, il devrait être possible de résoudre la plupart des problèmes usuels.

Au chapitre suivant, nous examinerons les caractéristiques précises des boîtiers d'interfaces d'entrées-sorties couramment connectés au 6502. Ensuite nous considérerons les structures de données fondamentales que le programmeur peut envisager d'utiliser.

EXERCICES

Exercice 6.21 : *Un afficheur à LED 7 segments peut aussi afficher des caractères autres que l'alphabet hexa. Calculez les codes pour : H, I, S, L, O, P, S, U, Y, g, h, i, j, l, n, o, p, r, t, u, y.*

Exercice 6.22

L'ordinogramme du traitement des interruptions apparaît sur la Figure 6-29 ci-dessous. Répondez aux questions suivantes :

a) Qu'est-ce qui est accompli par hardware, qu'est-ce qui est fait par software ?

b) A quoi sert le masque ?

c) Combien de registres faut-il préserver ?

d) Comment détermine-t-on le périphérique qui a causé l'interruption ?

e) Que fait l'instruction RTI ? Quelle est la différence avec le retour d'un sous-programme ?

f) Suggérer une manière de traiter une situation de débordement de la pile.

g) Quel est le « temps perdu » à cause du mécanisme d'interruption ?

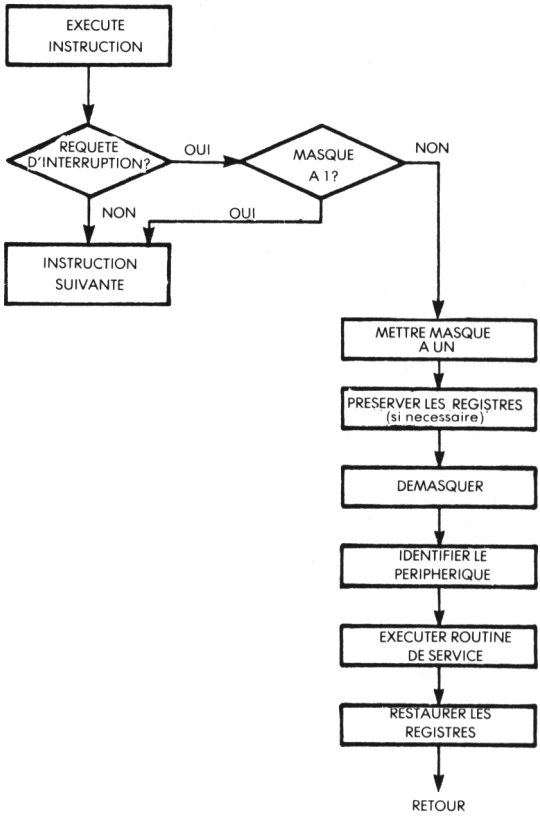


Fig. 6-29 : Logique des interruptions

7

COMPOSANTS D'ENTREES-SORTIES

INTRODUCTION

Nous avons appris à programmer le microprocesseur 6502 dans la plupart des situations courantes. Néanmoins, il faut faire une mention particulière des boîtiers d'entrées-sorties normalement connectés au microprocesseur. Grâce aux progrès de l'intégration des LSI, de nouveaux boîtiers sont apparus qui n'existaient pas auparavant. Il en résulte que programmer un système implique naturellement de programmer le microprocesseur lui-même, mais aussi de *programmer les boîtiers d'entrées-sorties.* En fait, il est souvent plus difficile de se rappeler comment programmer les différentes options de commande d'un boîtier d'entrées-sorties que de programmer le microprocesseur lui-même ! Ce n'est pas parce que la programmation est en elle-même plus difficile mais c'est parce que chacun de ces boîtiers a ses particularités propres. Nous allons examiner ici le boîtier d'entrées-sorties le plus général, le boîtier d'entrées-sorties programmable (le « PIO » = programmable input-output), puis nous verrons des « améliorations » de ce PIO standard, généralement utilisées avec le 6502 : les 6520, 6530, 6522 et 6532. Une étude complète des boîtiers d'entrée-sortie est présentée dans le volume suivant : « Applications du 6502 ».

Le PIO standard (6520)

Il n'y a pas de « PIO » standard. Cependant le 6520 est essentiellement analogue du point de vue fonctionnel à tous les PIO similaires produits dans le même but par les autres constructeurs. Le but d'un PIO est de fournir une connexion multiport pour les périphériques. (Un « port » est simplement un ensemble de 8 lignes d'entrées-sorties). Chaque PIO fournit au moins deux ensembles de lignes 8 bits

pour les unités d'entrées-sorties. Chaque unité nécessite un tampon de données au moins pour stabiliser à la sortie le contenu du bus de données. Notre PIO possèdera donc au moins un tampon pour chaque port.

En outre, nous avons établi que le microordinateur utilisera une procédure de *synchronisation des échanges* (handshaking), ou bien des *interruptions* pour communiquer avec le périphérique. Le PIO lui aussi, utilisera une procédure similaire pour communiquer avec le périphérique. Chaque PIO doit posséder au moins *deux lignes de commande par port* pour réaliser la fonction de synchronisation des échanges.

Le microprocesseur aura besoin aussi de lire l'état de chaque port. Chaque port doit donc être muni d'un ou plusieurs *bits d'état.*

Enfin, chaque PIO offre un certain nombre d'options pour configurer ses ressources. Le programmeur doit être capable d'accéder à un registre particulier à l'intérieur du PIO pour spécifier les options programmables. C'est le *registre de commande.* Dans le cas du 6520, l'information d'état fait partie du registre de commande.

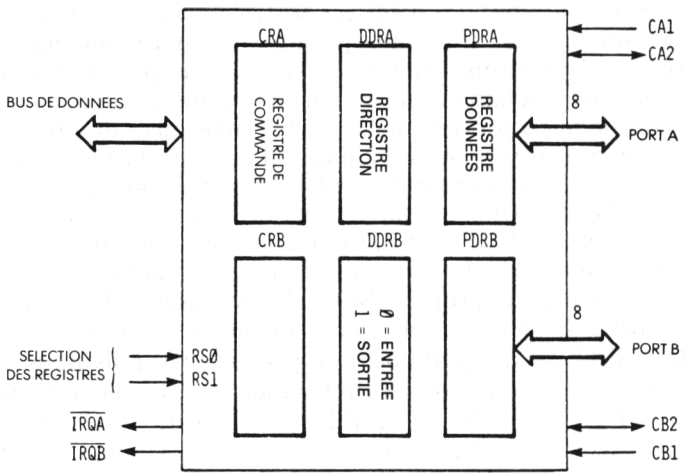


Fig. 7-1 : PIO typique

Une possibilité essentielle du PIO est que chaque ligne peut être configurée soit comme entrée soit comme sortie. Le diagramme d'un PIO apparaît Figure 7-1. Le programmeur peut spécifier si une ligne quelconque sera une entrée ou une sortie. Pour programmer la direction des lignes, un *registre de direction des données* est fourni pour chaque port. Un « 0 » sur un bit du registre de direction des

données spécifie une entrée. Un « 1 » spécifie une sortie. Il peut être surprenant que « 0 » soit utilisé pour « entrée » et « 1 » pour « sortie », alors que « 0 » devrait correspondre à sortie (« Output ») et « 1 » devrait correspondre à entrée (« Input »). Ceci est délibéré : lorsqu'on met le système sous tension, il est de la plus haute importance que toutes les lignes d'entrées-sorties soient configurées *en entrée.* Sinon, si le microordinateur est connecté à un périphérique dangereux, il risque de l'activer accidentellement. Quand on effectue un « RESET », tous les registres sont, en principe, remis à zéro, et cela a donc ainsi pour effet de configurer toutes les lignes périphériques du PIO en entrée. La liaison au microprocesseur apparaît sur la gauche de l'illustration. Naturellement, le PIO se relie aux 8 bits du bus de données, au bus d'adresse du microprocesseur et à son bus de commande. Le programmeur spécifie simplement l'adresse de n'importe quel registre auquel il souhaite accéder au sein du PIO. Le 6520, qui est compatible avec le Motorola 6820, hérite d'une de ses particularités : il possède 6 registres. Mais on ne peut spécifier qu'un registre parmi *quatre !* La façon dont ce problème est résolu est de commuter la valeur du bit 2 du registre de commande. Quand ce bit est à « 0 », on accède au registre direction des données correspondant. Quand il est à « 1 », on accède au registre de données. Par suite, quand le programmeur veut écrire des données dans le registre directement, il doit d'abord s'assurer que le bit 2 du registre de commande concerné est zéro, avant de pouvoir sélectionner ce registre. Ceci est quelquefois pénible à programmer, mais il est important de s'en souvenir sous peine de graves difficultés.

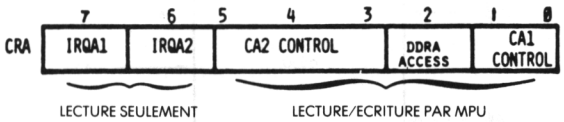


Fig. 7-2 : Format du mot de commande du PIA

RS1	RS0	CRA 2	CRB 2	REGISTER SELECTED
0	0	1	-	PERIPHERAL REGISTER A
0	0	0	-	DATA DIRECTION REGISTER A
0	1	-	-	CONTROL REGISTER A
1	0	-	1	PERIPHERAL REGISTER B
1	0	-	0	DATA DIRECTION REGISTER B
1	1	-	-	CONTROL REGISTER B

Fig. 7-3 : Adressage des registres d'un PIA

Pour clarifier la sélection des adresses dans le 6520, on consultera la table de la Figure 7-3. RS0 et RS1 sont deux signaux de sélection de registre, dérivés du bus d'adresse. En d'autres termes, ils représentent deux bits de l'adresse spécifiée par le programmeur. CRA est le registre de commande pour le port A. CRA (2) est le bit 2 de ce registre. CRB est le registre de commande pour le port B.

Le registre de commande interne

Le registre de commande du 6520 spécifie, comme nous l'avons vu, par son bit 2, un mode de sélection pour les registres internes du port. En outre, il fournit un certain nombre d'options pour générer ou tester des interruptions, ou pour effectuer des fonctions de « poignée de main » automatique. La description complète des facilités fournies n'est pas nécessaire ici. Simplement, l'utilisateur de tout système concret qui fait appel au 6520 devra se référer à la fiche technique qui montre l'effet produit par les différents bits du registre de commande (voir livre des « Applications du 6502 »). Chaque fois que le système est initialisé, le programmeur doit charger le registre de commande du 6520 avec le motif binaire convenable pour l'application envisagée.

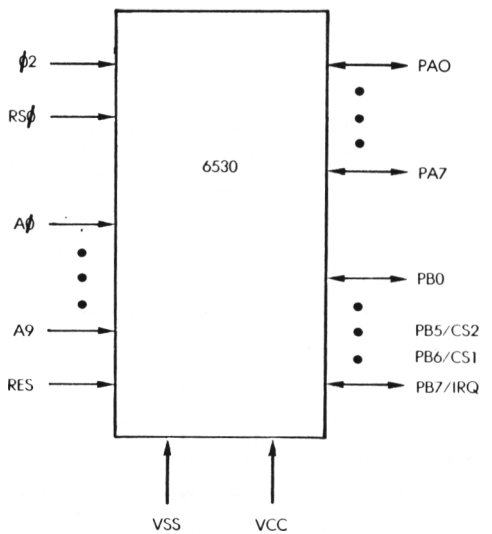


Fig. 7-4 : Le 6530

Le 6530

Le 6530 réalise une combinaison de quatre fonctions : RAM, ROM, PIO et temporisateur. La RAM est une mémoire 64 × 8. La ROM est une mémoire 1K × 8. Le temporisateur offre au programmeur de multiples possibilités de réalisation de délais internes. La partie PIO est, pour l'essentiel, analogue au 6520 que nous avons décrit : il y a deux ports, chaque port a un registre de données et un registre direction des données. Un « 0 » dans une position donnée du registre direction spécifie une entrée, tandis qu'un « 1 » spécifie une sortie.

Le générateur d'intervalles de temps programmable peut être programmé pour compter jusqu'à 256 intervalles (puisqu'il possède 8 bits).

Le programmeur peut spécifier une période de temps de 1, 8, 64 ou 1024 fois la période de l'horloge système. Quand le compte est épuisé, l'indicateur d'interruption du boîtier est mis à 1. Le contenu du temporisateur est établi par l'intermédiaire du bus de données. Les quatre intervalles de temps possibles doivent être spécifiés à l'aide des lignes A0 et A1 du bus adresse.

Trois broches du port B ont un double rôle : PB5, PB6 et PB7 peuvent être utilisées pour des fonctions de commande. La broche PB7, par exemple, peut être programmée comme broche de demande d'interruption.

Ce boîtier est utilisé, en particulier, sur la carte KIM (Remarque : sur le KIM, la broche PB6 n'est pas disponible comme entrée-sortie).

Programmation d'un PIO

A titre d'exemple, voici un programme d'utilisation d'un 6530 ou d'un 6522 (dont on suppose que le registre de commande est déjà positionné).

```
LDA   #$FF    PREPARER LE REGISTRE DIRECTION
STA   DDRB    CONFIGURER LE PORT B EN SORTIE
LDA   #$00
STA   IORB    GENERER UNE SORTIE ZERO
```

DDRB est l'adresse du registre direction des données du port B pour ce PIO. IORB est registre de donnée du port B. « FF » en hexadécimal est « 11111111 » en binaire d'où tous les bits en sortie.

Le 6522

Le 6522, appelé aussi VIA (« versatile interface adapter ») est une version améliorée du 6520. En plus des possibilités du 6520, il pos-

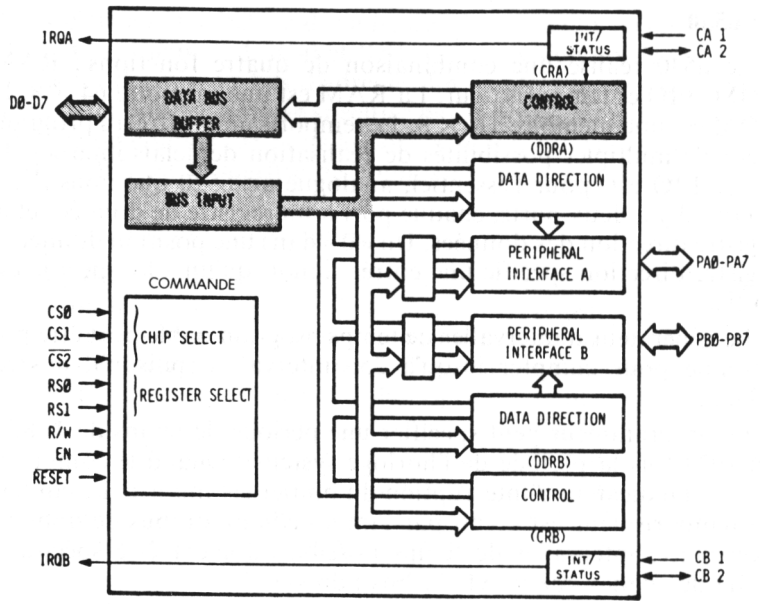


Fig. 7-5 : Utilisation d'un PIA : Chargement du registre de commande

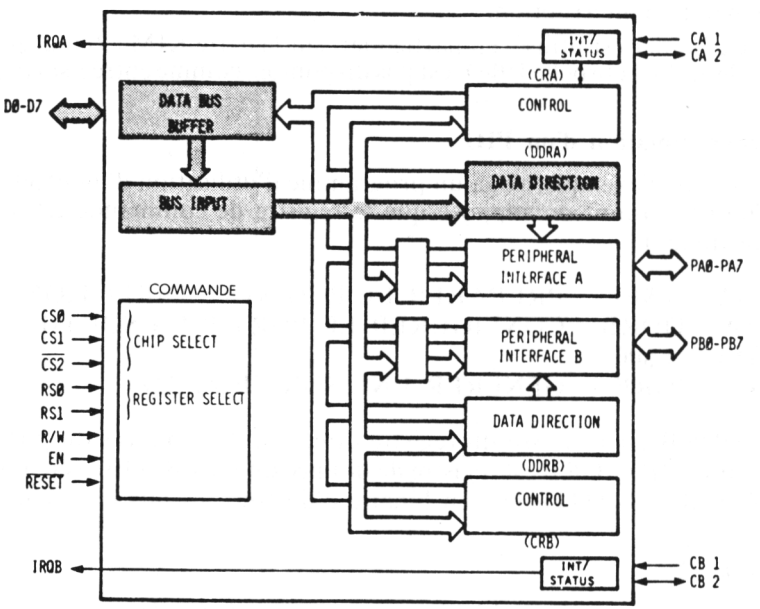


Fig. 7-6 : Utilisation d'un PIA : Chargement du registre direction

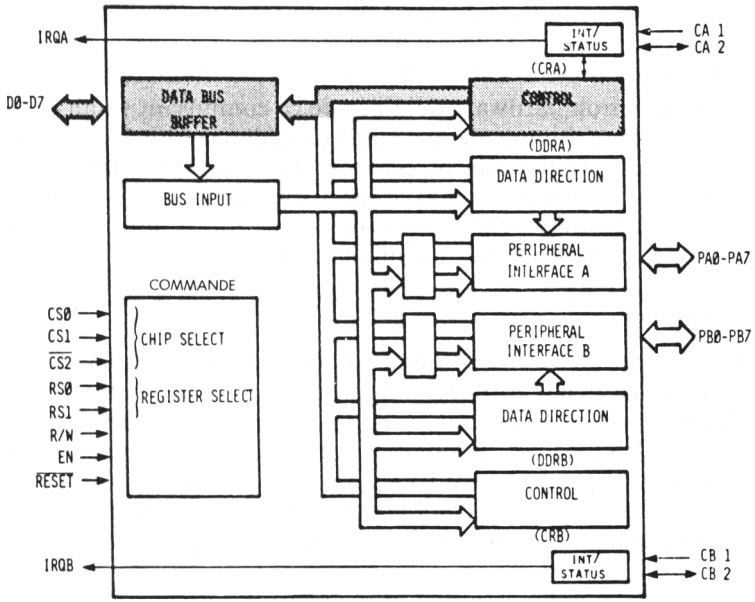


Fig. 7-7 : Utilisation d'un PIA : Lecture de l'état

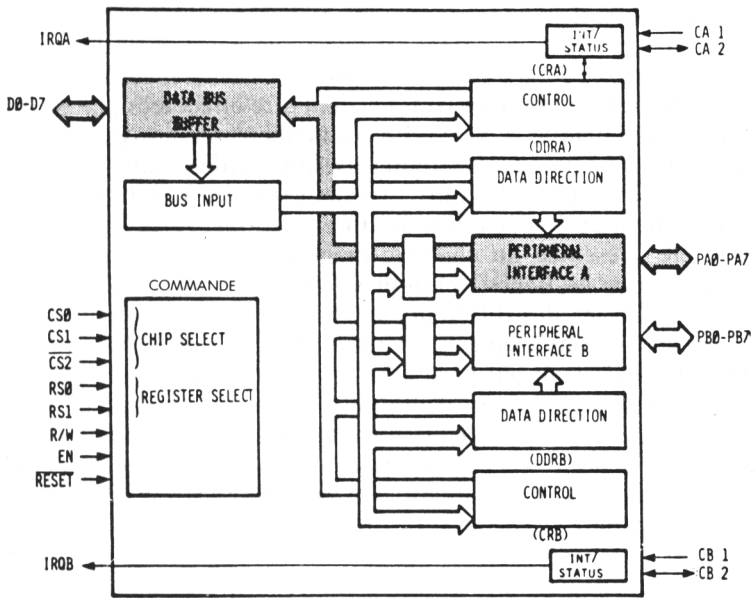


Fig. 7-8 : utilisation d'un PIA : Lecture des données d'entrée

sède deux générateurs d'intervalles de temps programmable, un convertisseur série-parallèle et parallèle-série, et la possibilité de verrouiller les données d'entrée sur front d'échantillonnage (latching).

La description hardware détaillée de ce composant va au-delà du sujet de ce livre. Simplement, à l'aide de la description qui a été donnée pour les composants précédents, il devrait être facile au programmeur de se familiariser avec l'adressage des registres internes de ce composant ainsi qu'avec sa programmation. Cette information est fournie dans les fiches techniques du constructeur.

Le 6532

Le 6532 est un boîtier combiné qui comprend une RAM 128×8, un PIO avec deux ports bidirectionnels et un générateur d'intervalles de temps programmable. Il est utilisé sur la carte SYM, fabriquée par SYNERTEK, analogue à la carte KIM fabriquée par MOS Technology et par ROCKWELL. Là encore, l'utilisateur doit examiner avec soin les fiches techniques de ce composant pour apprendre comment adresser et utiliser les différents registres internes.

RECAPITULATION

Malheureusement, pour utiliser effectivement de tels composants, il est nécessaire de comprendre en détail la fonction de chaque bit ou de chaque groupe de bits dans les différents registres de commande. Ces nouveaux boîtiers complexes automatisent un certain nombre de procédures, qui auparavant devaient être effectuées par software ou avec une logique spéciale. En particulier, une grande partie des procédures de « poignée de main » est automatisée dans des composants tels que le 6522. En outre, on peut trouver à l'intérieur un peu de traitement et de détection des interruptions. Avec l'information qui a été présentée au chapitre précédent, le lecteur devrait être capable de lire les fiches techniques correspondantes et de comprendre les fonctions des différents signaux et registres. Naturellement de nouveaux composants sont toujours sur le point d'être introduits : ils réaliseront par hardware des algorithmes encore plus complexes. Là aussi, le lecteur devrait être en mesure de les comprendre en étudiant avec soin les fiches techniques des constructeurs. A nouveau, pour le lecteur soucieux de comprendre de manière détaillée les entrées-sorties, la lecture du livre des « Applications du 6502 » est recommandée.

8

EXEMPLES D'APPLICATIONS

INTRODUCTION

Ce chapitre a pour but de tester vos nouvelles aptitudes à programmer, en présentant une collection de programmes utilitaires. Ces programmes ou « routines » se rencontrent fréquemment dans les applications habituelles : c'est pourquoi on les appelle généralement « programmes utilitaires ».

Ils nécessitent une synthèse des connaissances et des techniques présentées jusqu'ici.

Nous allons chercher des caractères à partir d'un organe d'entrées-sorties, et les traiter de différentes manières. Mais d'abord mettons à zéro une zone de la mémoire (cela n'est pas forcément nécessaire — chacun de ces programmes n'est présenté qu'à titre d'exemple de programmation).

MISE A ZERO D'UNE ZONE MEMOIRE

Nous voulons mettre à zéro le contenu de la mémoire, de l'adresse BASE + 1 à l'adresse BASE + LONGUEUR, où LONGUEUR est inférieure à 256.

Le programme est le suivant :

```
ZERO    LDX   # LONGUEUR
        LDA   # 0
NUL     STA   BASE,X
        DEX
        BNE   NUL
        RTS
```

Remarquez que le registre X est utilisé comme index pour pointer sur l'emplacement-mémoire courant à mettre à zéro.

L'accumulateur A est chargé une fois pour toutes avec la valeur « 0 » (tous les bits à « 0 »), puis écrit aux emplacements-mémoire successifs :

BASE + LONGUEUR, BASE + LONGUEUR — 1 etc. jusqu'à ce que X atteigne la valeur « 0 ». Quand X vaut « 0 », on effectue le retour du sous-programme.

Ce programme pourrait servir par exemple dans un test mémoire où on mettrait un bloc à « 0 », pour vérifier ensuite qu'on a bien « 0 » à la relecture.

Exercice 8.1 : *Ecrire un programme de test mémoire qui mette à zéro un bloc de 256 octets puis vérifie que chaque case est bien à « 0 ». Ensuite on écrit tous les bits à « 1 » et on vérifie le contenu du bloc. Puis on écrit « 01010101 » et on vérifie le contenu. Enfin, on écrit « 10101010 » et on vérifie.*

Maintenant, interrogeons nos périphériques pour déterminer lequel demande une intervention.

SCRUTATION DE PERIPHERIQUES

Nous supposerons qu'il y a 3 périphériques connectés à notre système. Leurs registres d'état sont aux adresses ETAT1, ETAT2, ETAT3.

Si leur bit d'état est en position 7, nous n'aurons qu'à lire le registre d'état et à tester son bit de signe. Si le bit d'état est à une autre position que 7, nous aurons avantage à utiliser l'instruction BIT du 6502 :

```
TEST    LDA    MASQUE
        BIT    ETAT1
        BNE    TROUVE1
        BIT    ETAT2
        BNE    TROUVE2
        BIT    ETAT3
        BNE    TROUVE3
        (sortie si non trouvé)
```

le masque contiendra, par exemple, « 00100000 » si nous testons la position 5. L'instruction BIT a pour résultat de mettre à « 0 » le bit Z si « MASQUE ET ETAT » est non nul c'est-à-dire si le bit corres-

pondant de ETAT est en accord avec le bit de MASQUE. L'instruction BNE (branchement si non égal à zéro) entraînera un branchement à la routine TROUVE appropriée.

LECTURE DE CARACTERES

Supposons que nous venons de trouver qu'un caractère est prêt au clavier. Accumulons les caractères dans une zone-mémoire appelée tampon jusqu'à ce que nous rencontrerons un caractère appelé SPC dont le code a été défini préalablement. Le sous-programme LIRECAR va chercher un caractère au clavier (cf. Chapitre 6 pour plus de détails) et le laisse dans l'accumulateur. Nous supposons qu'on cherche au maximum 256 caractères avant de trouver un caractère SPC.

```
                                INITIALISER L'INDEX A
CHAINE    LDX   #0              ZERO
SUIVANT   JSR   LIRECAR
                                EST-CE LE CARACTERE
          CMP   #SPC            D'ARRET
          BEQ   FIN             SI OUI FIN
                                SINON SAUVER LE
          STA   TAMPON,X        CARACTERE
                                INCREMENTER
          INX                   LE POINTEUR
                                PRENDRE
                                LE PROCHAIN
          JMP   SUIVANT         CARACTERE
FIN       RTS
```

Exercice 8.2 : *Améliorons la routine de base :*

a) renvoyer le caractère en écho (pour un Télétype par exemple) ;

b) vérifier que la chaîne entrée n'est pas plus longue que 256 caractères.

Nous avons maintenant une chaîne de caractères dans un tampon mémoire. Effectuons différents traitements sur elle.

TEST D'UN CARACTERE

Déterminons si le caractère situé à l'adresse LOC est égal à 0, 1 ou 2 :

```
Z12   LDA   LOC
      CMP   #$00
      BEQ   ZERO
      CMP   #$01
      BEQ   UN
      CMP   #$02
      BEQ   DEUX
      JMP   NON
```

Nous lisons simplement le caractère, puis nous utilisons l'instruction CMP pour vérifier sa valeur.

Effectuons maintenant un autre test.

TEST DANS UN INTERVALLE

Déterminons si le caractère ASCII à l'adresse LOC est un chiffre compris entre « 0 » et « 9 ».

```
INTERV  LDA   #$40      MISE A 1
        ADC   #$40      DU BIT V
        LDA   LOC
        CMP   #$30      ASCII 0
        BCC   INF
        CMP   #$39      ASCII 9
        BEQ   SORTIE    EGAL A 9
        BCS   SUP
SORTIE  CLC             MISE A 0 DE C
        CLV             MISE A 0 DE V
        RTS
INF     SEC             MISE A 1 DE C
        CLV
        RTS
SUP     RTS             C ET V SONT A 1
```

Le code ASCII de 0 est « 30 » en hexadécimal. Celui de 9 est « 39 ». Rappelez-vous que lorsqu'on utilise une instruction CMP, le bit de retenue sera mis à « 1 » si la valeur du littéral qui suit est inférieure ou égale au contenu de l'accumulateur. Il sera mis à « 0 » si le littéral est supérieur. Si « 30 » est supérieur au caractère, notre caractère est en-dessous de l'intervalle et il y a branchement.

Sinon, nous le comparons à « 39 » (9). S'il est inférieur ou égal à 9 tout est bien et nous sortons. Sinon nous allons en SUP.

Quand nous sortons de ce programme, nous voulons savoir si le caractère est en-dessous ou en-dessus du bien dans l'intervalle 0 à 9. Cela sera indiqué par les bits C et V. V n'est pas altéré par CMP qui change les indicateurs Z, N et C.

Lorsqu'on retourne de ce sous-programme, un « 1 » en C et V signifie « au-dessus » un « 1 » en C (et V = 0) signifie « au-dessous » et C = 0, V = 0 indique que le chiffre est correct, compris entre 0 et 9.

Naturellement, on pourrait utiliser d'autres conventions comme de charger un nombre dans l'accumulateur pour indiquer le résultat des tests.

Exercice 8.3 : *Simplifier le programme ci-dessus en le faisant tester par rapport au caractère ASCII qui suit « 9 » au lieu de tester par rapport à « 9 » lui-même.*

Exercice 8.4 : *Déterminer si un caractère ASCII contenu dans l'accumulateur est une lettre de l'alphabet.*

Lorsqu'on utilise une table ASCII, on remarque que l'on fait presque toujours appel à la parité. Par exemple le code ASCII de « 0 » est « 0110000 » (7-bits). Cependant si nous utilisons, par exemple, la parité impaire (nous garantissons que le nombre total de uns d'un mot soit impair) alors le code devient « 10110000 ». Un « 1 » supplémentaire est ajouté sur la gauche. Cela fait « B0 » en hexadécimal. Développons donc un programme pour générer la parité.

GENERATION DE PARITE

Ce programme génère une parité impaire en position 7

```
PARITE    LDX  #$07     COMPTE DES BITS
          LDA  #$00
          STA  CPT1     COMPTE DES 1
          LDA  CAR      LIRE LE CARACTERE
          ROL  A        ELIMINER LE BIT 7
SUIVANT   ROL  A        BIT SUIVANT
          BCC  ZERO     EST-CE UN 1 ?
UN        INC  CPT1
                        DECREMENTER
ZERO      DEX           LE COMPTE DES BITS
```

```
BNE   SUIVANT   DERNIER BIT?
ROL   A         RESTAURER LE BIT 0
ROL   A         ELIMINER LE BIT 7
                LE BIT LE PLUS A DROITE
LSR   CPTI      EST LA PARITE
ROR   A         LE METTRE EN A
RTS
```

On utilise le registre X pour compter les bits à mesure qu'ils sont décalés à gauche de l'accumulateur. A chaque fois qu'un « 1 » est décalé (il est testé par BCC) on incrémente le compteur de uns. Quand les 8 bits ont été décalés (le programme ignore le bit 7, qui deviendra le bit de parité), on décale A deux fois de plus pour ramener le bit 6 à gauche de A.

Le bit de parité voulu est le bit le plus à droite de CPTI : il est envoyé dans l'indicateur de retenue par LSR. Un autre LSR A le copie en position 7 de A et c'est fini.

Exercice 8.5 : *A l'aide du programme ci-dessous, vérifiez la parité d'un mot. Vous devez calculer la parité correcte puis la comparer à celle qui est fournie.*

CONVERSION DE CODE : ASCII vers DCB

La conversion de ASCII à DCB est très simple. Nous observerons que la représentation hexadécimale des caractères ASCII 0 à 9 est 30 à 39.

La représentation DCB s'obtient simplement en éliminant le « 3 » c'est-à-dire en masquant le quartet de gauche.

```
LDA   CAR
AND   #$0F    ELIMINATION DU QUARTET DE GAUCHE
STA   DCB
```

Exercice 8.6 : *Ecrire un programme pour convertir de DCB en ASCII.*

Exercice 8.7 : *(plus difficile) Ecrire un programme pour convertir de DCB en binaire.*

Conseil : $N_3\ N_2\ N_1\ N_0$ binaire vaut $(N_3 \times 10) + N_2) \times 10 + N_1) \times 10 + N_0$ en binaire.
Pour multiplier par 10, utiliser un décalage à gauche ($= \times 2$), un autre décalage à gauche ($= \times 4$), un ADC ($= \times 5$) un autre décalage à gauche ($= \times 10$).

En notation DCB complète, le premier mot peut contenir le nombre de chiffres DCB, le quartet suivant contient le signe et chaque quartet qui suit contient un chiffre DCB (nous supposons l'absence de décimales). Le dernier quartet du bloc peut être inutilisé.

TROUVER LE PLUS GRAND ELEMENT D'UNE TABLE

L'adresse de début de la table est contenue à l'adresse mémoire BASE en page-zéro. Le premier élément de la table est le nombre d'octets qui la composent. Ce programme cherche le plus grand élément de la table. La valeur sera laissée dans l'accumulateur A et sa position sera rangée dans la case-mémoire INDEX. Ce programme utilise les registres A et Y et fait appel à l'adressage indirect de sorte qu'il peut traiter n'importe quelle table n'importe où en mémoire.

```
MAX     LDY  #0          POINTEUR VERS LA TABLE
                         ACCES A L'ELEMENT 0
        LDA  (BASE),Y    = LONGUEUR
        TAY              LONGUEUR EN Y
                         INITIALISATION
        LDA  #0          A 0 DU MAXIMUM
                         INITIALISATION
        STA  INDEX       A 0 DE l'INDEX
                         L'ELEMENT COURANT EST-IL
BOUCLE  CMP  (BASE),Y    LE MAX
        BCS  PASCHG      OUI PAS DE CHANGEMENT
        LDA  (BASE),Y    CHARGER LE NOUVEAU MAX
        STY  INDEX       EMPLACEMENT DU MAX
                         POINTER VERS L'ELEMENT
PASCHG  DEY              SUIVANT
        BNE  BOUCLE      FAUT-IL CONTINUER
        RTS              FINI SI Y = 0
```

Le programme teste d'abord le nième élément. S'il est plus grand que 0 il va en A et son rang est mémorisé dans INDEX. Puis on teste le (N-1) ième élément, etc.

Le programme fonctionne pour des entiers positifs.

Exercice 8.8 : *Modifier le programme afin qu'il fonctionne aussi pour des nombres négatifs en complément à 2.*

Exercice 8.9 : *Ce programme marche-t-il aussi pour des caractères ASCII ?*

Exercice 8.10 : *Ecrire un programme qui classe n noms (de 3 caractères chacun) par ordre alphabétique.*

Exercice 8.11 : *Ecrire un programme qui classe N nombres en ordre croissant.*

SOMME DE N ELEMENTS

Ce programme calcule la somme sur 16 bits de N éléments d'une table. L'adresse de début de la table est contenue dans la case-mémoire BASE en page-zéro. Le premier élément de la table contient le nombre d'éléments N. La somme sur 16 bits sera laissée dans les cases-mémoire SOMBAS et SOMHTE. Si la somme nécessitait plus de 16 bits, on ne garderait que les 16 bits les plus bas (on dit que les bits les plus significatifs sont tronqués).

Le programme modifie les registres A et Y. Il suppose un maximum de 256 éléments.

```
        LDA   #0          INITIALISATION
        STA   SOMBAS      A ZERO
        STA   SOMHTE      DE LA SOMME
        TAY               ET DE Y
        LDA   (BASE),Y    PRENDRE N
        TAY               LE METTRE EN Y
        CLC               C A O POUR ADC
                          PRENDRE LE PROCHAIN
BOUCLE  LDA   (BASE),Y    ELEMENT
        ADC   SOMBAS      L'AJOUTER A SOMBAS
        STA   SOMBAS      SAUVER LE RESULTAT
        BCC   PASRET      RETENUE ?
        INC   SOMHTE      L'AJOUTER A SOMHTE
                          POUR LA PROCHAINE
        CLC               ADDITION
                          PASSER AU PROCHAIN
PASRET  DEX               ELEMENT
        BNE   BOUCLE      CONTINUER SI Y NON NUL
        RTS
```

Ce programme est simple et devrait être clair.

Exercice 8.12 : *Modifier ce programme pour calculer :*
a) une somme sur 24-bits ;
b) une somme sur 32-bits ;
c) pour détecter tout débordement.

CALCUL D'UNE SOMME DE CONTRÔLE

Une somme de contrôle (checksum) est un chiffre ou un ensemble de chiffres calculé à partir d'un bloc de caractères successifs. La somme de contrôle est calculée au moment où on range les données, et elle est placée à leur suite. Pour vérifier l'intégrité des données, la somme de contrôle est recalculée lorsqu'on lit les données et elle est comparée à la valeur stockée. Tout désaccord dénote une erreur ou une panne.

On utilise différents algorithmes. Ici, nous allons faire le ou exclusif de tous les octets d'une table de N éléments et laisser le résultat dans l'accumulateur. Comme d'habitude la base de la table est stockée à l'adresse BASE en page zéro. Le premier élément de la table est le nombre d'éléments N. N doit être inférieur à 256. Le programme altère A et Y.

```
SOMCTL  LDY #0          POINTE VERS PREMIER
                        ELEMENT
        LDA (BASE),Y    RECUPERE N
        TAY             LE MET EN Y
        LDA #0          INITIALISE LA SOMME
                        DE CONTROLE
BCTL    EOR (BASE),Y    OU EXCLUSIF AVEC
                        PROCHAIN ELEMENT
        DEY             POINTE VERS LE
                        SUIVANT
        BNE BCTL        CONTINUE
        RTS
```

COMPTE DES ZEROS

Le programme compte combien il y a de zéros dans notre tableau habituel (adresse en ADR) et il laisse le nombre dans le registre X.

Il modifie A, X et Y :

```
ZEROS   LDY #0          POINTE VERS PREMIER
                        ELEMENT
        LDA (ADR),Y     RECUPERE N
        TAY             LE MET EN Y
        LDX #0          INITIALISE LE NOMBRE
                        DE ZEROS
BCL     LDA (ADR),Y     PROCHAIN ELEMENT
        BNE NONZ        EST-IL NUL ?
        INX             OUI : ON LE COMPTE
NONZ    DEY             POINTE VERS LE
                        SUIVANT
        BNE BCL         CONTINUE
        RTS
```

Exercice 8.13 : *Modifier ce programme pour compter : a) le nombre d'étoiles (caractère « * ») ; b) le nombre de lettres de l'alphabet ; c) le nombre de chiffres compris entre 0 et 9.*

RECHERCHE DANS UNE CHAINE DE CARACTERES

Une chaîne de caractères est rangée en mémoire comme le montre la Figure 8-1. Nous allons parcourir la chaîne jusqu'à l'apparition d'une sous-chaîne, ou motif (TEMPLT) de longueur TPTLEN. La longueur de la chaîne de départ est STRLEN et le programme retourne avec dans le registre X, l'emplacement où on a trouvé TEMPLT ou FF si le motif ne s'y trouve pas. L'ordinogramme correspondant au programme apparaît Figure 8-2. La chaîne est d'abord parcourue jusqu'à ce qu'on trouve le premier caractère du motif TEMPLT. Si on ne le trouve jamais le programme se termine sur un échec. Si le premier caractère est trouvé, on compare le second caractère du motif avec le caractère suivant de la chaîne. S'il ne concorde pas on recommence à chercher le premier caractère du

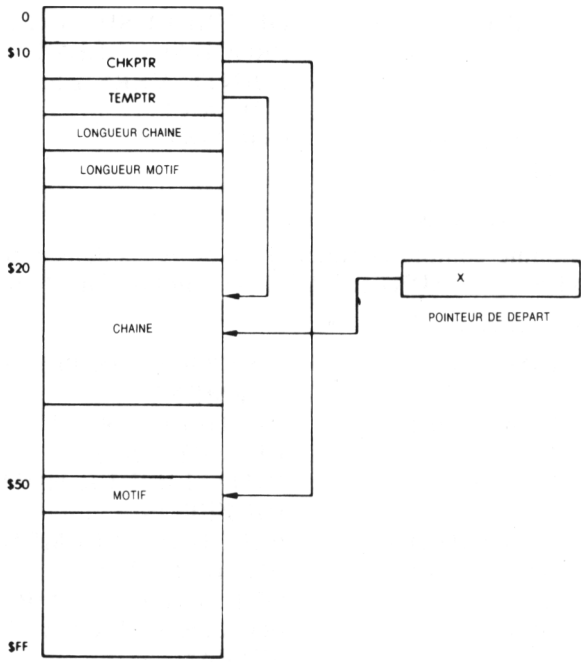


Fig. 8-1 : Recherche dans une chaîne : la mémoire

motif puisqu'il peut y en avoir une autre occurrence dans la chaîne donnée. Si le premier et le deuxième caractère concordent, on compare les suivants de la même manière. Le programme apparaît Figure 8-3. Notez que le registre X sert de pointeur vers l'élément courant de la chaîne. Naturellement, on utilise l'adressage indexé pour retrouver l'élément courant dans la chaîne.

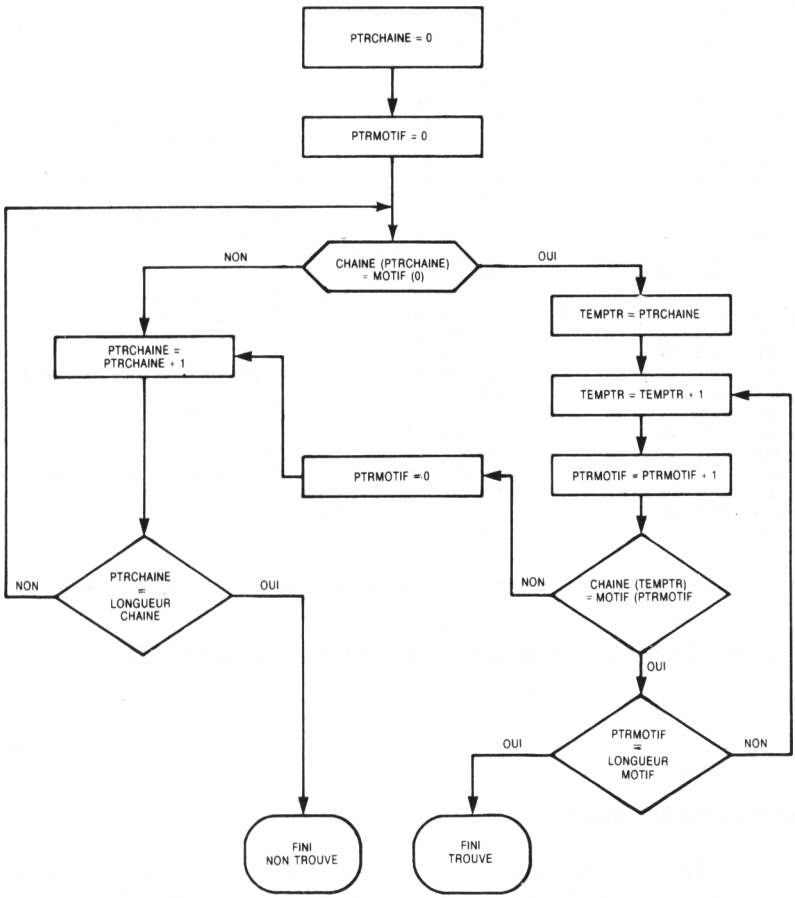


Fig. 8-2 : Ordinogramme de la recherche dans une chaîne

```
LINE # LOC     CODE       LINE

0002  0000              ;STRING SEARCH.
0003  0000              ;FINDS LOCATION IN STRING OF LENGTH 'STRLEN'
0004  0000              ;STARTING AT 'STRING' OF A TEMPLATE OF
0005  0000              ;LENGTH 'TPTLEN' STARTING AT 'TEMPLT', AND
0006  0000              ;RETURNS WITH X=LOCATION OF TEMPLATE
0007  0000              ;IN STRING IF FOUND, OR X=$FF IF NOT FOUND.
0008  0000              ;
0009  0000              STRING = $20          ;1ST LOCATION OF STRING.
0010  0000              TEMPLT = $50          ;1ST LOCATION OF TEMPLATE.
0011  0000                     * = $10
0012  0010              CHKPTR *=*+1
0013  0011              TEMPTR *=*+1
0014  0012              STRLEN *=*+1          ;LENGTH OF STRING.
0015  0013              TPTLEN *=*+1          ;LENGTH OF TEMPLATE.
0016  0014                     * = $200
0017  0200  A2 00        NXTPOS LDX #0         ;RESET SEARCH START POINTER.
0018  0202  A5 50               LDA TEMPLT    ;IS FIRST ELEMENT OF TEMPLATE...
0019  0204  D5 20               CMP STRING,X  ;= CURRENT STRING ELEMENT?
0020  0206  F0 08               BEQ CHECK     ;IF YES, CHECK FOR REST OF MATCH.
0021  0208  E8        NXTSTR INX            ;INCREMENT SEARCH START COUNTER.
0022  0209  E4 12               CPX STRLEN    ;IS IT EQUAL TO STRING LENGTH?
0023  020B  D0 F5               BNE NXTPOS    ;NO, CHECK NEXT STRING POSITION.
0024  020D  A2 FF               LDX #$FF      ;YES, SET 'NOT FOUND' INDICATOR.
0025  020F  60                  RTS           ;RETURN: ALL CHRS CHECKED.
0026  0210  86 11        CHECK  STX TEMPTR    ;LET TEMPORARY POINTER=
0027  0212              ;CURRENT STRING POINTER.
0028  0212  A9 00               LDA #0
0029  0214  85 10               STA CHKPTR    ;RESET TEMPLATE POINTER.
0030  0216  E6 11        CHKLP  INC TEMPTR    ;INCREMENT TEMPORARY POINTER.
0031  0218  E6 10               INC CHKPTR    ;INCREMENT TEMPLATE POINTER.
0032  021A  A4 10               LDY CHKPTR
0033  021C  C4 13               CPY TPTLEN    ;DOES TEMPLATE POINTER=TEMPLATE LENGTH?
0034  021E  F0 0C               BEQ FOUND     ;IF YES, TEMPLATE MATCHED.
0035  0220  B9 50 00            LDA TEMPLT,Y  ;LOAD TEMPLATE ELEMENT.
0036  0223  A4 11               LDY TEMPTR
0037  0225  D9 20 00            CMP STRING,Y  ;COMPARE TO STRING CHR.
0038  0228  D0 DE               BNE NXTSTR    ;IF NO MATCH, CHECK NEXT STRING CHR.
0039  022A  F0 EA               BEQ CHKLP     ;IF MATCH, CHECK NEXT CHR.
0040  022C  60          FOUND  RTS           ;DONE.
0041  022D                     .END
```

Fig. 8-3 : Programme de la recherche dans une chaîne

N.B. STRING = CHAINE ; CHKPTR = PTRMOTIF.

RECAPITULATION

On a dans ce chapitre présenté des routines utilitaires qui font appel à des combinaisons des techniques présentées dans les chapitres précédents. Les routines devraient maintenant vous permettre de commencer à écrire vos propres programmes. Beaucoup d'entre elles ont porté sur une structure de données particulière : la table. Mais il existe d'autres structures de données : on va maintenant les passer en revue.

9

STRUCTURES DE DONNEES

1re PARTIE-CONCEPTS

INTRODUCTION

La conception d'un bon programme comporte deux tâches : le développement d'un algorithme et le choix de structures de données. Dans la plupart des programmes simples, il n'intervient pas de structure de données spéciale, et le problème principal à surmonter pour apprendre la programmation est de s'entraîner à concevoir des algorithmes et à les coder de façon efficace dans un langage machine donné. C'est ce que nous avons fait jusqu'ici. Néanmoins, le développement de programmes plus complexes exige aussi la compréhension des structures de données. Deux structures de données ont déjà été utilisées partout dans ce livre : la table et la pile. Le but de ce chapitre est de présenter d'autres structures plus générales que vous pouvez souhaiter utiliser. Ce chapitre est complètement indépendant du microprocesseur ou même de l'ordinateur choisi. Il est théorique et concerne l'organisation logique des données dans le système. Il existe des livres spécialisés sur les structures de données, tout comme il en existe sur le sujet des multiplications efficaces, des divisions ou d'autres algorithmes habituels. Ce chapitre a donc été inclus pour rendre ce livre complet, mais il se limitera à l'essentiel. Il ne prétend pas être exhaustif. Les structures de données les plus fréquentes vont maintenant être passées en revue.

POINTEURS

Un pointeur est un nombre utilisé pour désigner l'emplacement de la donnée proprement dite. Tout pointeur est une adresse. Mais toute adresse n'est pas nécessairement appelée pointeur. Une adresse

n'est un pointeur que si elle pointe vers un certain type de données ou vers une information structurée. Nous avons déjà rencontré un pointeur typique : le pointeur de pile qui pointe vers le sommet de la pile (ou généralement, juste au-dessus de la pile). Nous verrons que la pile est une structure commune, appelée structure LIFO. L'adressage indirect en fournit un autre exemple : l'adresse indirecte est toujours un pointeur vers la donnée que l'on souhaite retrouver.

Exercice 9.1 : *Examinez la Figure 9-1. A l'adresse 15 de la mémoire, se touve un pointeur vers la table T. La table T commence à l'adresse 500. Quel est le contenu exact du pointeur vers T?*

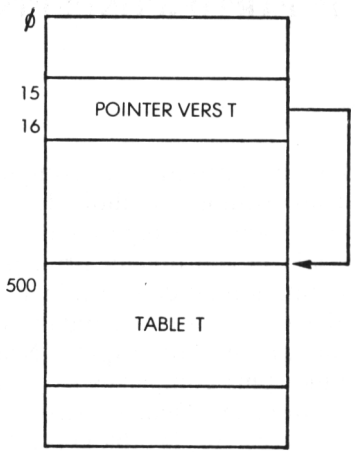


Fig. 9-1 : Pointeur d'indirection

LISTES

Presque toutes les structures de données sont organisées en listes de diverses sortes.

Listes séquentielles

La liste séquentielle, ou table, ou bloc, est probablement la structure de données la plus simple, et nous l'avons déjà utilisée. Les tables sont généralement ordonnées en fonction d'un critère spécifique tel que, par exemple, l'ordre alphabétique, ou l'ordre numérique. Il est alors facile de retrouver un élément dans la table, en se servant, par exemple, de l'adressage indexé, comme nous l'avons

fait. Un bloc désigne généralement un groupe de données qui a des limites définies mais dont le contenu n'est pas ordonné. Il peut, par exemple, contenir une chaîne de caractères. Ou il peut représenter un secteur sur un disque. Ou il peut former une certaine zone logique (appelée segment) de la mémoire. Dans des cas de ce genre, il peut être difficile d'accéder à un élément aléatoire dans le bloc.

Pour faciliter l'accès à des blocs d'information, on utilise des répertoires.

Répertoire

Un répertoire est une liste de tables ou de blocs. Par exemple, le système de fichiers utilise normalement une structure de répertoire. A titre d'exemple simple, le répertoire principal du système peut contenir une liste de noms des utilisateurs. Ceci est illustré sur la Figure 9-2. L'entrée correspondant à l'utilisateur « JOHN » pointe vers le répertoire des fichiers de JOHN. Le répertoire des fichiers est une table qui contient les noms de tous les fichiers de JOHN, et leur localisation. C'est là encore une table de pointeurs. Dans ce cas nous venons de constituer un répertoire à deux niveaux. Un système de répertoires souple permettra d'inclure les répertoires intermédiaires supplémentaires que l'utilisateur pourra juger convenables.

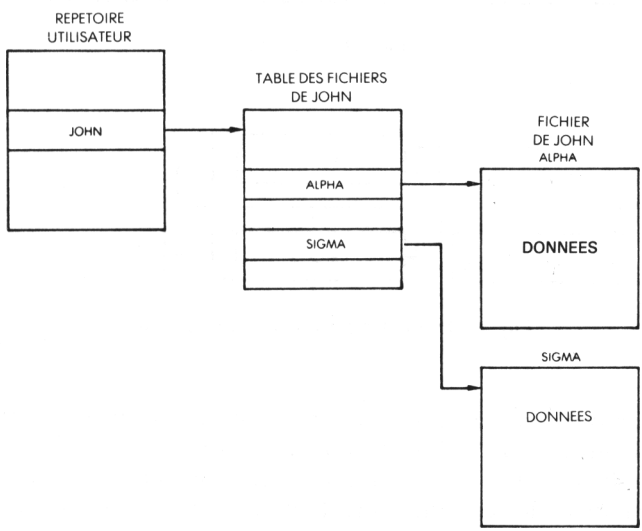


Fig. 9-2 : Structure d'un répertoire

Liste

Il y a souvent dans un système des blocs d'information qui représentent des données, ou des événements, ou d'autres structures qui ne peuvent pas être déplacés facilement. S'ils pouvaient être facilement déplacés, on les grouperait probablement dans une table pour les trier ou les structurer. Le problème est que nous voulons les laisser là où ils sont, et quand même les ordonner comme premier, second, troisième, quatrième. Ce problème est résolu par la structure de liste. Le concept de liste est illustré sur la Figure 9-3. Sur l'illustration, nous voyons qu'un pointeur de liste appelé PREMBLOC pointe vers le début du premier bloc. Un emplacement réservé dans le bloc 1, qui peut être, par exemple, le premier ou le dernier mot contient un pointeur vers le bloc 2, appelé PTR1. Le processus est répété pour le bloc 2 et le bloc 3. Comme le bloc 3 est le dernier élément de la liste, PTR3 contient par convention une valeur spéciale « nul » ou bien pointe vers lui-même pour que l'on puisse détecter la fin de la liste. Cette structure est économique vu qu'elle ne demande qu'un petit nombre de pointeurs (un par bloc) et permet à l'utilisateur d'éviter de déplacer les blocs en mémoire.

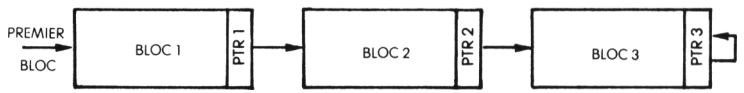


Fig. 9-3 : Liste

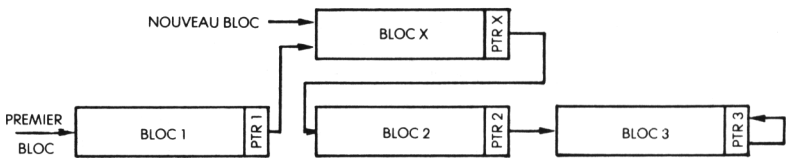


Fig. 9-4 : Insertion d'un nouveau bloc

Examinons, par exemple, comment on insère un nouveau bloc. Ceci est illustré sur la Figure 9-4. Supposons que le nouveau bloc soit à l'adresse NOUVEAU et doive être inséré entre les blocs 1 et 2. Le pointeur PTR1 est simplement changé afin de contenir la valeur NOUVEAU pour pointer maintenant vers le bloc X. PTRX va contenir l'ancienne valeur de PTR1, c'est-à-dire, qu'il va pointer vers le bloc 2. Les autres pointeurs de la structure sont inchangés.

Nous pouvons voir que l'insertion du nouveau bloc a simplement nécessité la mise à jour de deux pointeurs de la structure. Il est évident que c'est efficace.

Exercice 9.2 : *Dessiner le schéma montrant comment on enlèverait le bloc 2 de cette structure.*

Plusieurs types de listes ont été développés pour faciliter des types d'accès, d'insertions ou de suppressions spécifiques. Examinons quelques-uns des types de listes les plus fréquemment utilisés.

File

La file est appelée généralement liste FIFO (first in, first out : premier entré, premier sorti). Une file est montrée sur la Figure 9-5. Pour clarifier le schéma, nous pouvons supposer, par exemple, que le bloc sur la gauche est une routine de service d'un organe de sortie comme une imprimante. Les blocs qui apparaissent sur la droite sont les blocs de requête d'impression de caractères émanant de divers programmes ou routines. L'ordre suivant lequel ils seront servis est l'ordre établi par la file d'attente. On peut voir que le prochain événement qui va être servi est le bloc 1, le suivant est le bloc 2, et le suivant est bloc 3. Dans une file, la convention est que tout nouvel élément qui arrive dans la file est inséré à la fin. Ici, il serait inséré après PTR3. Cela garantit que le premier bloc qui avait été inséré dans la file sera le premier servi. Il est très fréquent dans un

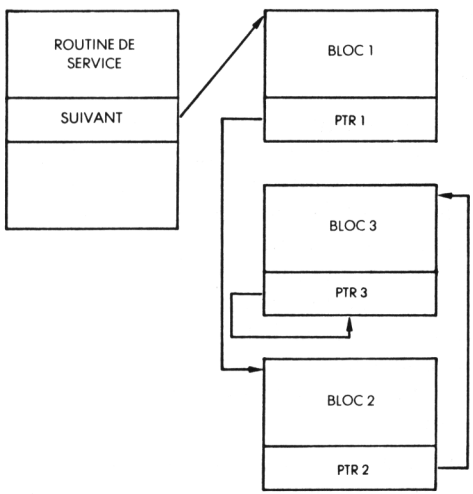


Fig. 9-5 : File

système d'ordinateur, d'avoir des files d'attente pour un certain nombre d'événements, à chaque fois qu'ils doivent attendre après une ressource rare comme le processeur ou un périphérique.

Pile

La structure de pile a déjà été étudiée en détail dans le livre. C'est une structure LIFO (last in, first out : dernier entré, premier sorti).

Le dernier élément déposé au sommet est le premier qui sera enlevé. Une pile peut être réalisée sous la forme d'un bloc ordonné, ou bien sous la forme d'une liste.

Comme la plupart des piles des microprocesseurs servent à des événements rapides comme les appels de sous-programmes ou les interruptions, on alloue généralement à la pile un bloc contigu plutôt que d'utiliser une liste.

Liste ou bloc ?

De même, la file pourrait être implantée dans un bloc de cases-mémoire réservées. L'avantage d'un bloc contigu est l'accès rapide et l'élimination des pointeurs. L'inconvénient est qu'il est souvent nécessaire de consacrer un bloc assez vaste pour accommoder la taille correspondant au cas le plus défavorable. En outre, il devient difficile d'insérer ou de supprimer des éléments à l'intérieur du bloc. Comme la mémoire est traditionnellement une ressource rare, les blocs ont traditionnellement été réservés aux structures de taille fixe, ou alors aux structures demandant la vitesse d'accès maximum, comme la pile.

Liste circulaire

La liste circulaire est une liste où le dernier élément repointe vers le premier. Ceci est illustré sur la Figure 9-6. Dans le cas d'une liste circulaire, on conserve souvent un pointeur sur le bloc courant. Dans le cas d'événements ou de programmes en attente de service, le pointeur sur l'événement courant sera déplacé d'un rang vers la droite ou la gauche à chaque fois. La liste circulaire correspond généralement à une structure où tous les blocs sont supposés avoir la même priorité. Toutefois, la liste circulaire peut être aussi utilisée comme cas particulier des autres structures simplement pour faciliter la récupération du premier bloc après le dernier lorsqu'on effectue une recherche.

Comme exemple de liste circulaire, un programme de scrutation utilise une liste circulaire pour interroger tous les périphériques puis revenir au premier.

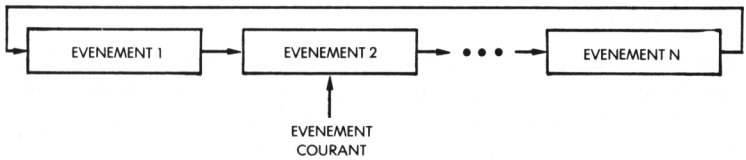


Fig. 9-6 : Liste circulaire

Arbres

Chaque fois qu'il existe une relation logique entre tous les éléments d'une structure (cela s'appelle d'habitude une syntaxe), on peut utiliser une structure arborescente. Un exemple simple en est l'arbre des descendants, ou arbre généalogique. C'est illustré sur la Figure 9-7. On peut voir que Simon a deux enfants : un fils Robert et une fille Jeanne.

Jeanne a, à son tour, trois enfants : Lise, Thomas et Philippe. Thomas a à son tour deux enfants : Max et Christian. En revanche, Robert (à gauche) n'a pas de descendants.

C'est un arbre structuré. Nous avons, en fait, déjà rencontré un exemple d'arbre simple sur la Figure 9-2. La structure de répertoire est un arbre à deux niveaux.

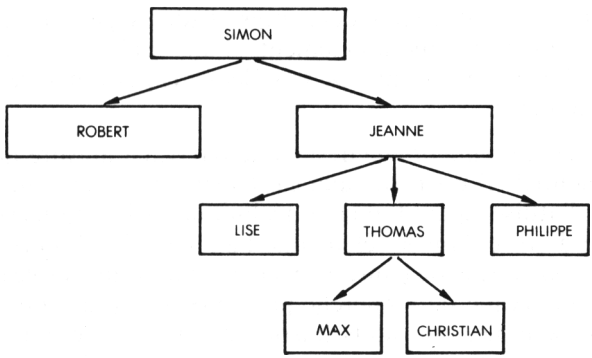


Fig. 9-7 : Arbre généalogique

On a avantage à utiliser un arbre à chaque fois que les éléments peuvent être classés en fonction d'une structure fixe. Cela facilite l'insertion et la récupération des éléments. En outre, ils peuvent former des groupes d'informations de façon structurée. Ceux-ci peuvent être nécessaires pour des traitements ultérieurs, comme dans la conception d'un compilateur ou d'un interpréteur.

Listes doubles

On peut établir des relations supplémentaires entre les éléments d'une liste. L'exemple le plus simple est la liste double, illustré Figure 9-8. On peut voir qu'on a la séquence habituelle de liens de gauche à droite, plus une autre suite de liaisons de droite à gauche. Le but est de permettre l'accès facile à l'élément précédent celui qu'on est en train de traiter, aussi bien qu'à lé'élément suivant.

Cela se fait au prix d'un pointeur supplémentaire par bloc.

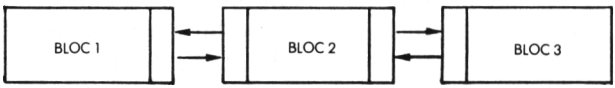


Fig. 9-8 : Liste double

RECHERCHE ET CLASSEMENT

La façon de classer les éléments d'une liste dépend directement du type de la structure utilisée. Beaucoup d'algorithmes de recherche ont été développés pour les structures de données les plus fréquemment utilisées. Nous avons déjà utilisé l'adressage indexé. Ceci est possible chaque fois que les éléments d'une table sont ordonnés en fonction d'un critère connu. De tels éléments peuvent être retrouvés par leur numéro.

La recherche séquentielle désigne le balayage linéaire de tout un bloc séquentiel. Il est clair, que c'est inefficace, mais on peut être obligé de l'utiliser quand aucune technique meilleure n'est disponible, faute d'ordre dans les éléments. La recherche *dichotomique* ou *logarithmique* essaie de trouver un élément dans une table classée en divisant l'intervalle de recherche par deux à chaque étape. Supposant, par exemple, que nous faisons une recherche dans une liste alphabétique, nous pourrions commencer par le milieu de la table et déterminer si le nom que nous cherchons est avant ou après cet endroit. S'il est après, nous éliminons la première moitié de la table et regardons l'élément situé au milieu de la seconde moitié. Nous comparons à nouveau cet élément au nom cherché et nous restreignons notre recherche à l'une des deux moitiés et ainsi de suite. La longueur maximum de la recherche est alors garantie être en Log n, où n est le nombre d'éléments de la table.

Il existe beaucoup d'autres techniques de recherche.

RECAPITULATION

Cette section n'avait pour but que de donner une brève présentation des structures de données habituelles qui peuvent être utilisées par un programmeur.

Bien que la plupart des structures de données habituelles aient été classées en types et aient reçu un nom, l'organisation globale des données dans un système complexe peut faire appel à toute combinaison de ces structures, ou demander au programmeur d'inventer des structures mieux adaptées. L'ensemble des possibilités n'est limité que par l'imagination du programmeur. De même, un grand nombre de techniques de recherche et de classement ont été développées pour traiter les structures de données habituelles. Une description complète sort du cadre de ce livre. Le contenu de cette section avait pour but d'insister sur l'importance de concevoir des structures appropriées pour les données à manipuler et de fournir les outils de base pour cette conception.

STRUCTURES DE DONNEES

2e PARTIE-EXEMPLES

INTRODUCTION

Nous allons présenter ici des exemples de programmes véritables, portant sur des structures de données type : table, liste, arbre. Nous programmerons des algorithmes pratiques de tri, de recherche et d'insertion pour ces structures, ainsi que des techniques avancées comme le hachage et la fusion.

Le lecteur intéressé par ces techniques de programmation est invité à analyser en détail les programmes présentés dans cette section. En revanche, le programmeur débutant pourra la survoler, en première lecture, et y revenir, ultérieurement lorsqu'il se sentira prêt.

Une bonne compréhension des concepts présentés dans la première partie de ce chapitre est nécessaire pour tirer profit des exemples. En outre, les programmes feront appel à tous les modes d'adressage du 6502, et intégreront la plupart des concepts et des techniques présentés dans les chapitres précédents. Nous allons maintenant introduire quatre structures : la table pure et simple, la

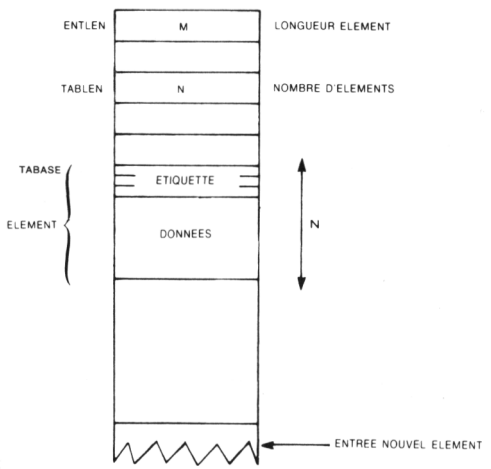


Fig. 9-9 : Structure de table

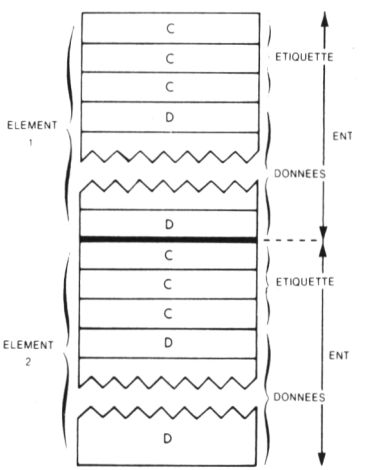


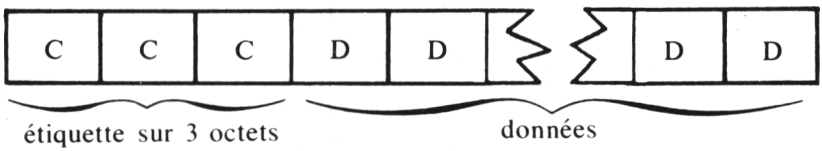
Fig. 9-10 : Eléments d'une table typique en mémoire

table alphabétique, la liste avec répertoire et l'arbre. Pour chaque structure, nous développerons trois programmes : recherche, ajout et suppression.

Trois algorithmes spécialisés seront décrits séparément à la fin de cette section : le hachage, le tri à bulle et la fusion.

REPRESENTATION DES DONNEES DANS LA TABLE

La table simple et la table alphabétique font appel à la même représentation, pour chaque élément :



Chaque élément comprend un label de 3 octets, et un bloc de n octets de données proprement dites, avec n compris entre 1 et 253. Chaque élément occupe ainsi, au maximum, une page (256 octets). A l'intérieur de chaque table, tous les éléments ont la même longueur (cf. Figure 9-10). Les programmes qui manipulent ces tables utilisent les conventions suivantes, sur les noms des variables :

ENTLEN est la longueur d'un élément. Par exemple, si chaque élément comprend 10 octets de données, ENTLEN = 3 + 10 = 13 octets.
TABASE est la base de la table en mémoire.
POINTR est un pointeur vers l'élément courant.
OBJECT est l'élément à ajouter ou supprimer.
TABLEN est le nombre d'éléments.

Tous les labels sont supposés distincts. La convention contraire n'entraînerait que des changements mineurs dans les programmes.

TABLE SIMPLE

La table simple est organisée comme une suite de n éléments, non-classés (cf. Figure 9-11).

Lors d'une recherche, toute la table doit être parcourue, jusqu'à la découverte d'un élément correspondant à ce qu'on cherche, ou jusqu'à la fin de la table. Les nouveaux éléments sont ajoutés derrière les éléments existants, au moyen d'une insertion. Lorsqu'un élément est supprimé, il est nécessaire de décaler les éléments d'adresse plus élevée, pour conserver la continuité de la table.

Recherche

Nous utiliserons une technique de recherche linéaire, consistant à comparer, lettre par lettre, le contenu du champ étiquette de chaque élément de la table, à son tour, avec l'étiquette de l'élément cherché, OBJECT.

Le pointeur courant, POINTR, est initialisé à la valeur TABASE. Le registre index X est initialisé au nombre d'éléments de la table (à l'adresse TABLEN).

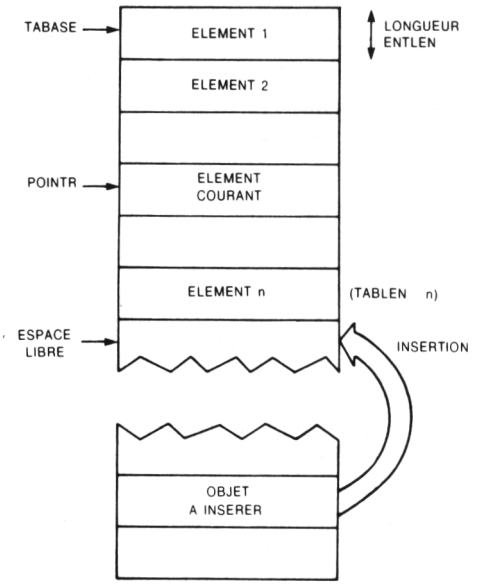


Fig. 9-11 : Table simple

La recherche se fait de façon évidente. L'ordinogramme correspondant apparaît à la Figure 9-12, et le programme à la Figure 9-16, à la fin de cette section (programme « SEARCH »).

Insertion d'un élément

L'insertion d'un nouvel élément nécessite le recours au premier bloc mémoire de (ENTLEN) octets, disponible à la fin de la table (cf. Figure 9-11).

Le programme vérifie d'abord que le nouvel élément ne figure pas déjà dans la table (nous supposerons, dans cet exemple, que toutes les étiquettes doivent être différentes). Si tel n'est pas le cas, il

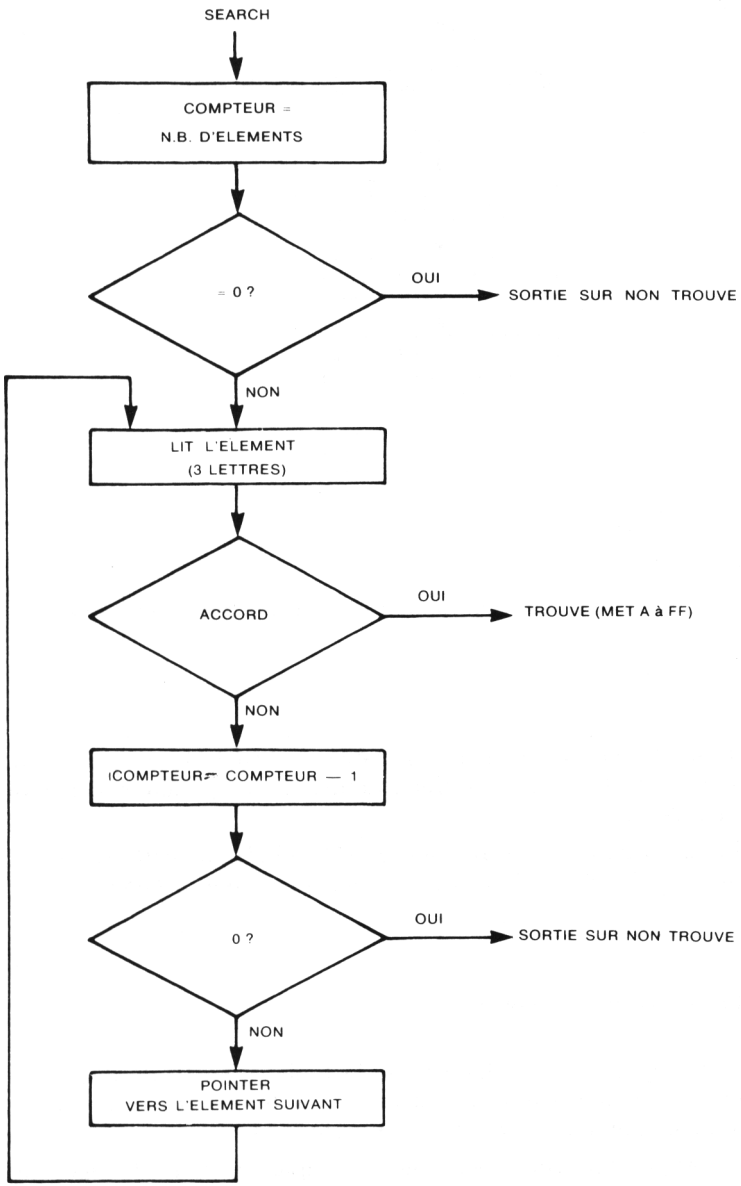


Fig. 9-12 : Ordinogramme de recherche dans une table

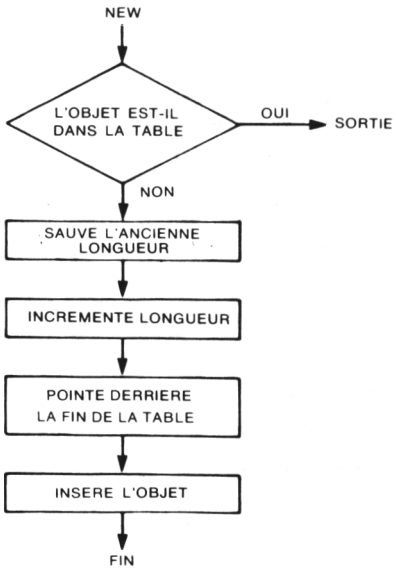


Fig. 9-13 : Ordinogramme d'insertion dans une table

incrémente la longueur de la table, TABLEN, et place l'objet OBJECT à la fin. L'ordinogramme correspondant est montré Figure 9-13.

Le programme apparaît Figure 9-16, à la fin de cette section. Son nom est « NEW ». Il réside aux adresses 0636 à 0659.

Suppression d'un élément

Pour supprimer un élément de la table, il suffit de décaler d'un cran les éléments qui le suivent. La longueur de la table est diminuée, comme le montre la Figure 9-14.

Le programme correspondant est simple. Il apparaît, Figure 9-16, sous le nom de « DELETE », et réside aux adresses 0659 à 0686. L'ordinogramme correspondant est montré Figure 9-15.

La variable TEMPTR sert de pointeur temporaire, pointant vers l'élément à remonter.

Le registre index Y est mis à la longueur d'un élément de la table : il sert aux transferts de blocs. Notez l'utilisation de l'adressage indirect indexé :

```
(0672)  LOOPE   DEY
                LDA     (TEMPTR), Y
                STA     (POINTR), Y
                CPY     #0
                BNE     LOOPE
```

Lors du transfert, POINTR pointe toujours vers le « trou », c'est-à-dire vers la cible du prochain transfert de bloc.

L'indicateur Z sert à marquer la réussite de la suppression, lors de la sortie.

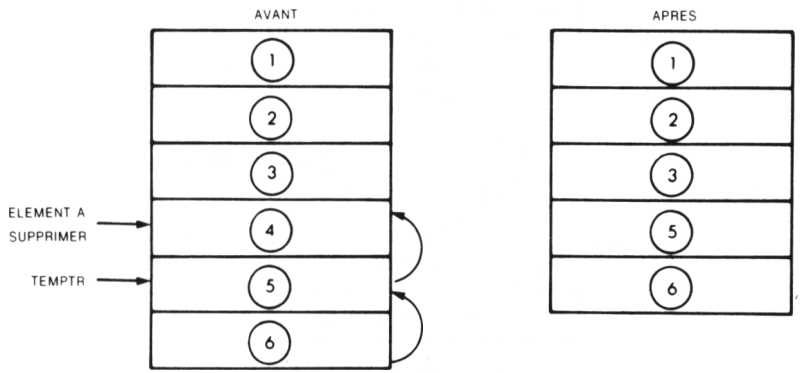


Fig. 9-14 : Suppression d'un élément (table simple)

LISTE ALPHABETIQUE

Au contraire de la structure précédente, dans la liste (ou tableau) alphabétique, tous les éléments sont conservés, classés par ordre alphabétique, ce qui permet d'utiliser des techniques de recherche plus rapides que la recherche linéaire. Nous nous servirons, ici, de la recherche dichotomique.

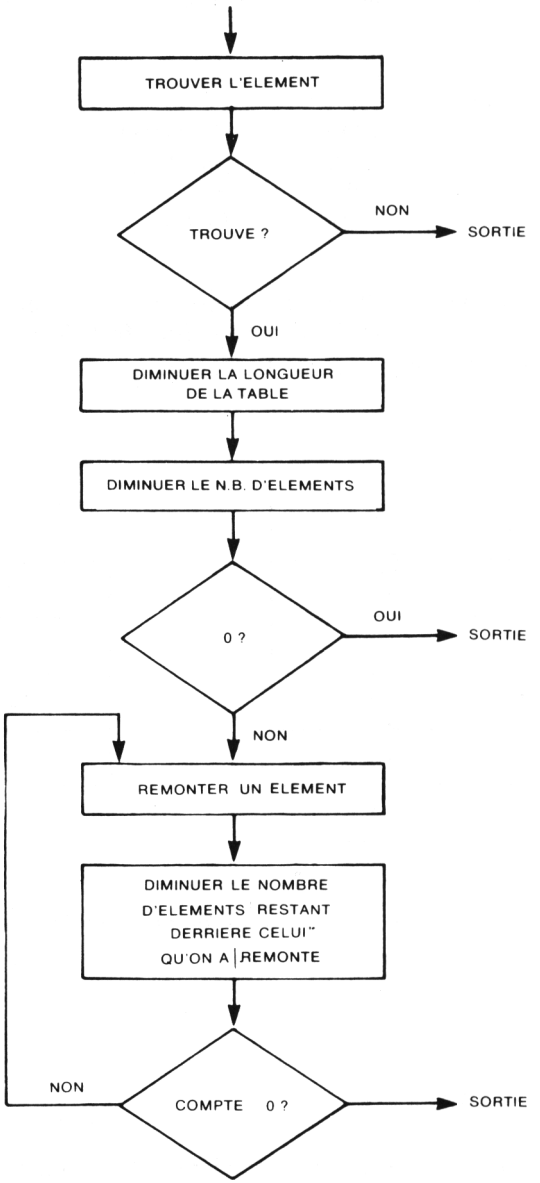


Fig. 9-15 : Ordinogramme de la suppression d'un élément dans une table

Recherche

L'algorithme est celui de la recherche dichotomique classique. Rappelons que la technique est, pour l'essentiel, semblable à celle qu'on utilise pour trouver un nom dans un annuaire. On commence la recherche quelque part au milieu de l'annuaire, et selon les noms trouvés, on se dirige vers l'avant ou vers l'arrière pour découvrir le nom recherché. Cette méthode est rapide et, somme toute, assez simple à réaliser.

L'ordinogramme de recherche dichotomique est montré Figure 9-17. Le programme apparaît Figure 9-22.

Il conserve les éléments par ordre alphabétique, et les retrouve par recherche dichotomique (ou « logarithmique »). Un exemple apparaît Figure 9-18.

```
LINE # LOC     CODE       LINE

0002  0000              TABASE = $10
0003  0000              POINTR = $12
0004  0000              TABLEN = $14
0005  0000              OBJECT = $15
0006  0000              ENTLEN = $17
0007  0000              TEMPTR = $18
0008  0000              ;
0009  0000                    *=$600
0010  0600              ;
0011  0600  A5 10       SEARCH LDA TABASE      ;INITIALIZE POINTER
0012  0602  85 12              STA POINTR
0013  0604  A5 11              LDA TABASE+1
0014  0606  85 13              STA POINTR+1
0015  0608  A6 14              LDX TABLEN       ;STORE TABLEN AS A VARIABLE
0016  060A  F0 29              BEQ OUT          ;CHECK FOR 0 TABLE
0017  060C  A0 00       ENTRY  LDY #0           ;COMPARE FIRST LETTERS
0018  060E  B1 15              LDA (OBJECT),Y
0019  0610  D1 12              CMP (POINTR),Y
0020  0612  D0 0E              BNE NOGOOD
0021  0614  C8                 INY              ;COMPARE SECOND LETTERS
0022  0615  B1 15              LDA (OBJECT),Y
0023  0617  D1 12              CMP (POINTR),Y
0024  0619  D0 07              BNE NOGOOD
0025  061B  C8                 INY              ;COMPARE THIRD LETTERS
0026  061C  B1 15              LDA (OBJECT),Y
0027  061E  D1 12              CMP (POINTR),Y
0028  0620  F0 11              BEQ FOUND
0029  0622  CA          NOGOOD DEX              ;SEE HOW MANY ENTRIES ARE LEFT
0030  0623  F0 10              BEQ OUT
0031  0625  A5 17              LDA ENTLEN       ;ADD ENTLEN TO POINTER
0032  0627  18                 CLC
0033  0628  65 12              ADC POINTR
0034  062A  85 12              STA POINTR
0035  062C  90 DE              BCC ENTRY
0036  062E  E6 13              INC POINTR+1
0037  0630  4C 0C 06           JMP ENTRY
0038  0633  A9 FF       FOUND  LDA #$FF         ;CLEAR Z FLAG IF FOUND
```

Fig. 9-16 : Programme concernant une table simple : SEARCH (Recherche), NEW (Insertion), et DELETE (Suppression)

```
0039  0635  60          OUT    RTS
0040  0636              ;
0041  0636              ;
0042  0636              ;
0043  0636  20 00 06    NEW    JSR SEARCH       ;SEE IF OBJECT IS THERE
0044  0639  D0 1D              BNE OUTE
0045  063B  A6 14              LDX TABLEN       ;CHECK FOR 0 TABLE
0046  063D  F0 0B              BEQ INSERT
0047  063F  A5 12              LDA POINTR       ;POINTER IS AT LAST ENTRY
0048  0641  18                 CLC              ;..MUST MOVE IT TO END OF TABLE
0049  0642  65 17              ADC ENTLEN
0050  0644  85 12              STA POINTR
0051  0646  90 02              BCC INSERT
0052  0648  E6 13              INC POINTR+1
0053  064A  E6 14    INSERT INC TABLEN       ;INCREMENT TABLE LENGTH
0054  064C  A0 00              LDY #0           ;MOVE OBJECT TO END OF TABLE
0055  064E  A6 17              LDX ENTLEN
0056  0650  B1 15    LOOP   LDA (OBJECT),Y
0057  0652  91 12              STA (POINTR),Y
0058  0654  C8                 INY
0059  0655  CA                 DEX
0060  0656  D0 F8              BNE LOOP
0061  0658  60          OUTE   RTS              ;Z SET IF WAS DONE
0062  0659              ;
0063  0659              ;
0064  0659              ;
0065  0659  20 00 06    DELETE JSR SEARCH       ;FIND WHERE OBJECT IS
0066  065C  F0 2D              BEQ OUTS         ;EXIT IF NOT FOUND
0067  065E  C6 14              DEC TABLEN       ;DECREMENT TABLE LENGTH
0068  0660  CA                 DEX              ;SEE HOW MANY ENTRIES ARE
0069  0661  F0 26              BEQ DONE         ;..AFTER ONE TO BE DELETED
0070  0663  A5 12    ADDEN  LDA POINTR       ;ADD ENTLEN TO POINTER AND
0071  0665  18                 CLC              ;..SAVE AT TEMP STORAGE
0072  0666  65 17              ADC ENTLEN
0073  0668  85 18              STA TEMPTR
0074  066A  A9 00              LDA #0
0075  066C  65 13              ADC POINTR+1     ;ADD CARRY TO HIGH BYTE
0076  066E  85 19              STA TEMPTR+1
0077  0670  A4 17              LDY ENTLEN
0078  0672  88          LOOPE  DEY
0079  0673  B1 18              LDA (TEMPTR),Y   ;SHIFT ONE ENTRY OF MEMORY DOWN
0080  0675  91 12              STA (POINTR),Y
0081  0677  C0 00              CPY #0
0082  0679  D0 F7              BNE LOOPE
0083  067B  CA                 DEX              ;DECREMENT ENTRY COUNTER
0084  067C  F0 0B              BEQ DONE
0085  067E  A5 18              LDA TEMPTR       ;MOVE TEMP TO POINTER
0086  0680  85 12              STA POINTR
0087  0682  A5 19              LDA TEMPTR+1
0088  0684  85 13              STA POINTR+1
0089  0686  4C 63 06           JMP ADDEN
0090  0689  A9 FF    DONE   LDA #$FF         ;CLEAR Z FLAG IF IT WAS DONE
0091  068B  60          OUTS   RTS
0092  068C              ;
0093  068C              ;
0094  068C                     .END

ERRORS = 0000 <0000>

ADDEN   0663   DELETE   0659   DONE    068B   ENTLEN   0017
ENTRY   060C   FOUND    0633   INSERT  064A   LOOP     0650
LOOPE   0672   NEW      0636   NOGOOD  0622   OBJECT   0015
OUT     0635   OUTE     0658   OUTS    068D   POINTR   0012
SEARCH  0600   TABASE   0010   TABLEN  0014   TEMPTR   0018

END OF ASSEMBLY
```

Fig. 9-16 : Programme concernant une table simple : SEARCH (Recherche), NEW (Insertion), et DELETE (Suppression) (suite)

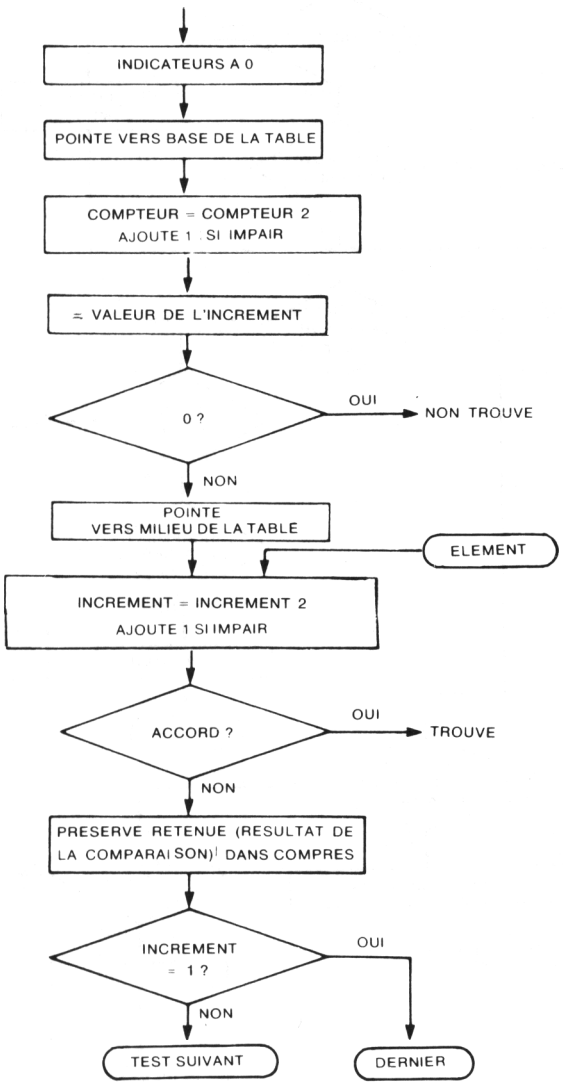


Fig. 9-17 : Ordinogramme de la recherche dichotomique

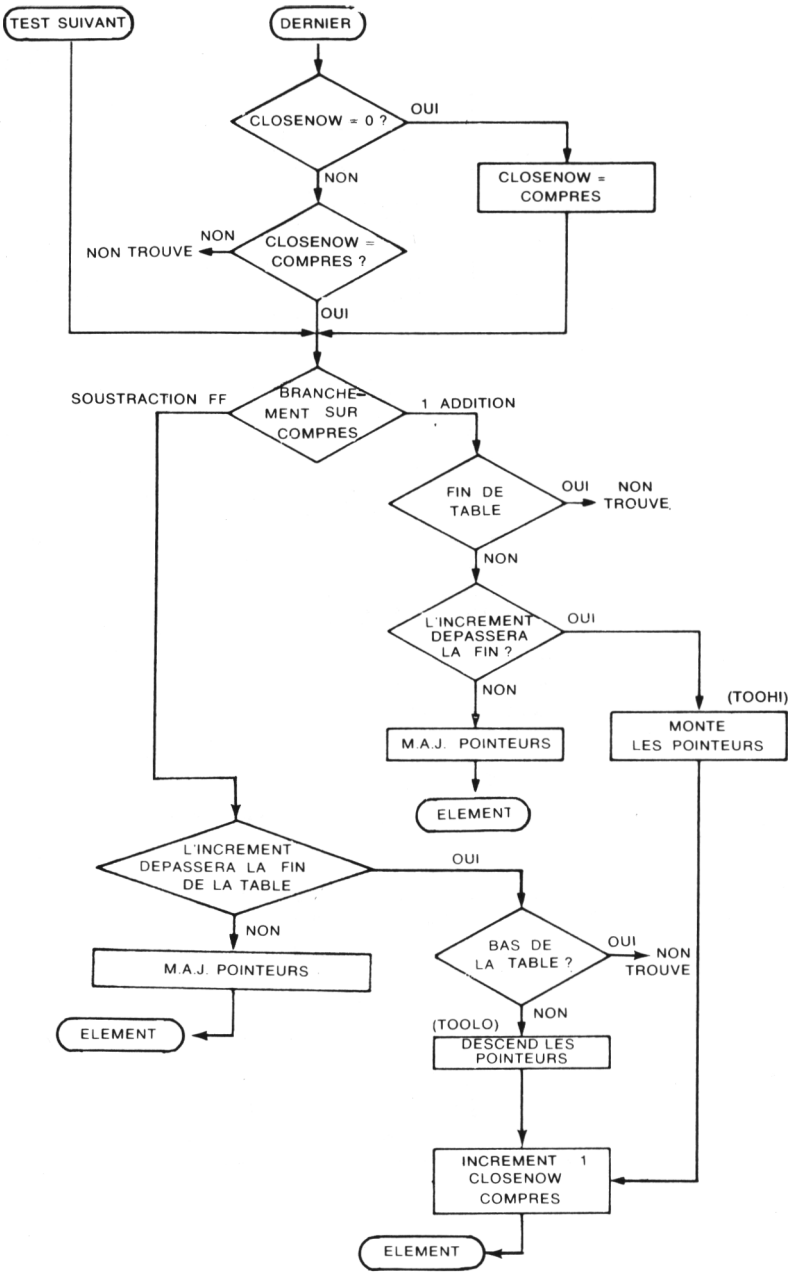


Fig. 9-17 : Ordinogramme de la recherche dichotomique (suite)

La recherche est quelque peu compliquée par la nécessité de garder trace de plusieurs conditions. Le problème principal est d'éviter de rechercher un objet absent, car dans ce cas nous ne cesserions pas de tester, alternativement, les valeurs immédiatement inférieure et supérieure alphabétiquement. Pour nous épargner cette peine, nous conserverons, dans le programme, la valeur de l'indicateur de retenue résultant de l'échec d'une comparaison (dans la variable COMPRS). Lorsque la valeur INCMNT, qui précise le niveau d'incrémentation du pointeur, atteint la valeur 1, un autre indicateur, appelé CLOSE, reçoit la valeur de COMPRS. Ainsi, dans la mesure où tous les incréments futurs seront 1, si le pointeur dépasse l'emplacement supposé de l'objet recherché, CMPRS ne sera plus égal à CLOSE, et la recherche s'arrêtera. Cette technique permet à la routine NEW de déterminer l'emplacement des pointeurs, physique et logique, vers l'emplacement où ira l'objet.

Ainsi, si l'objet recherché n'est pas dans la table, et si le pointeur courant est incrémenté de 1, l'indicateur CLOSE sera positionné. Lors de la passe suivante, le résultat de la comparaison sera l'opposé du précédent. Les deux indicateurs seront en désaccord, et le programme sortira en indiquant « non trouvé ».

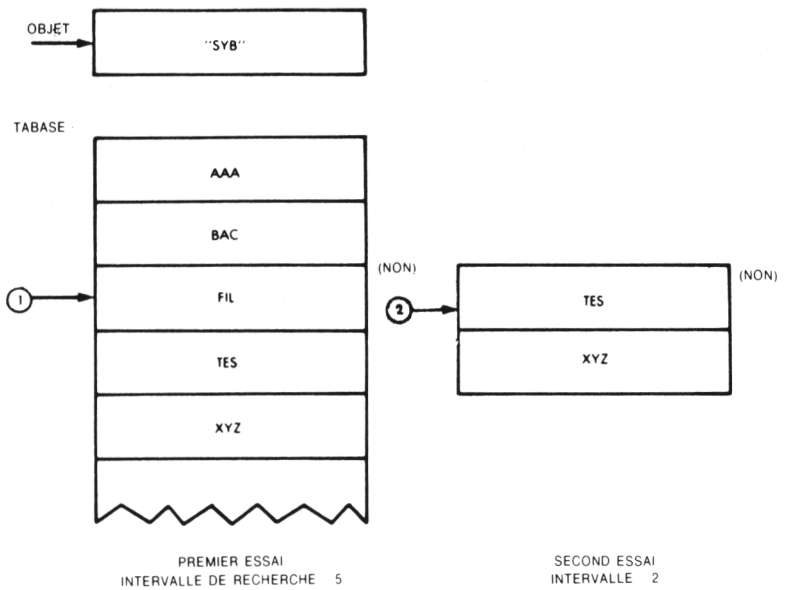


Fig. 9-18 : Recherche dichotomique

Second problème majeur : la possibilité de sortir de la table lorsqu'on ajoute, ou soustrait, l'incrément. La solution passe par la mise en œuvre d'un test d'addition, ou de soustraction, utilisant un pointeur logique et la valeur de sa longueur, pour déterminer le nombre effectif d'éléments, plutôt que des pointeurs physiques, pour déterminer, simplement, leur position.

En résumé, le programme utilise deux indicateurs pour mémoriser des informations : CMPRES et CLOSE. L'indicateur CMPRES sert à mémoriser le fait que la retenue était à 0 ou à 1, après la comparaison la plus récente. Cela nous permet de déterminer si l'élément testé est plus petit ou plus grand que celui auquel on le compare. Si la retenue C est à 1, l'élément est plus petit que l'objet cherché, et CMPRES est mis à 1. Si la retenue C est à 0, l'élément est plus grand que l'objet, et CMPRES est mis à « FF ».

Notez aussi que, lorsque la retenue est à 1 (non trouvé), le pointeur courant pointe vers l'élément situé en-dessous de l'objet.

Le second indicateur est CLOSE. Il est égal à CMPRES lorsque l'incrément INCMNT devient égal à 1, et il détecte l'échec de la recherche de l'élément : CMPRES ne sera pas égal à CLOSE au prochain tour.

Les autres variables utilisées par le programme sont : LOGPOS, qui indique la position logique dans la table (numéro de l'élément), INCMNT, qui représente la valeur dont le pointeur courant sera incrémenté, ou décrémenté, si la prochaine comparaison ne donne pas l'égalité. TABLEN représente, comme d'habitude, la longueur totale de la liste. LOGPOS et INCMNT sont comparés à TABLEN, pour s'assurer que les limites de la table ne seront pas dépassées. Le programme, appelé « SEARCH », apparaît Figure 9-22.

Il réside entre les adresses 0600 et 06E3, et mérite d'être étudié avec soin, car il est beaucoup plus complexe que celui de la recherche linéaire. Une complication supplémentaire tient au fait que l'intervalle de recherche peut être pair ou impair. Dans le premier cas, il sera nécessaire d'apporter une correction : on ne peut pas, par exemple, pointer vers le milieu d'une table de quatre éléments.

Dans le second cas, on a recours à un « truc » pour pointer vers l'élément du milieu. La division par 2 est effectuée par décalage à droite. Le bit qui tombe dans l'indicateur de retenue, après l'instruction LSR, sera 1 si l'intervalle était pair. Il est simplement ajouté à nouveau, au pointeur.

```
(0615)  DIV     LSR     A       DIVISE PAR 2
                ADC     #0      RAJOUTE LA RETENUE
                STA     LOGPOS  NOUVEAU POINTEUR
```

L'objet cherché est alors comparé avec l'élément, au milieu de l'intervalle de recherche. Si la comparaison donne l'égalité, on sort, sinon (NOGOOD), la retenue est mise à 0, si l'objet est inférieur à l'élément. Lorsque INCMNT devient 1, on vérifie que l'indicateur CLOSE (initialisé à 0) était bien à 1. Dans le cas contraire, on l'y mettra, mais il faudra alors effectuer un test pour déterminer si l'emplacement où devait se trouver l'objet a été dépassé.

Insertion d'un élément

Pour insérer un nouvel élément, il est nécessaire d'effectuer une recherche dichotomique. Si l'élément est dans la table, il n'y a pas lieu de l'insérer (tous les éléments sont supposés distincts). Si l'élément n'a pas été trouvé dans la table, il doit être inséré. La valeur de CMPRES, après la recherche, indique si l'élément doit être inséré immédiatement, après, ou juste avant, le dernier élément de la table auquel il a été comparé. Tous les éléments qui suivent l'emplacement de la future insertion sont alors décalés d'un cran, et le nouvel élément est inséré.

Le processus d'insertion est illustré Figure 9-19. Le programme correspondant apparaît Figure 9-22.

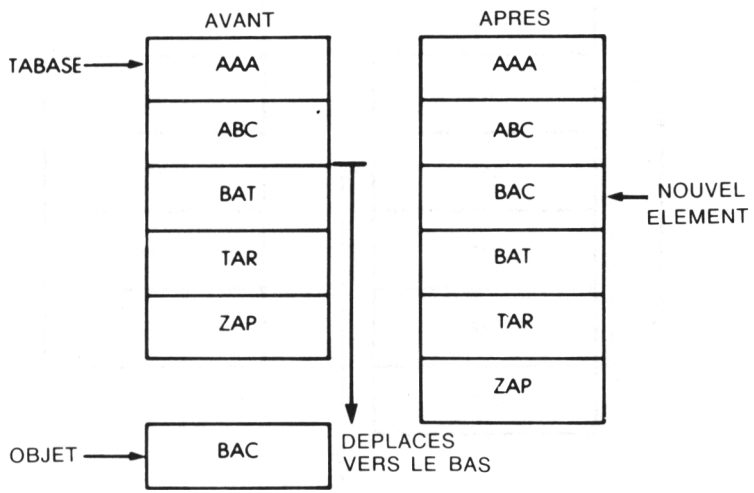


Fig. 9-19 : Insertion de « BAC »

Il est appelé « NEW », et réside aux adresses 06E3 à 075E. Notez que nous utilisons, là encore, l'adressage indirect pour les transferts de blocs :

```
(072A)          LDY   ENTLEN
        ANOTHR  DEY
                LDA   (POINTR), Y
                STA   (TEMP), Y
                CPY   #0
                BNE   ANOTHR
```

Même chose à l'adresse 0750.

Suppression d'un élément

De même, nous effectuerons une recherche dichotomique pour découvrir un élément à supprimer. Si la recherche n'aboutit pas, il n'y aura pas de suppression. Si l'élément est trouvé, il sera supprimé, en décalant vers le haut tous les éléments qui lui font suite. La Figure 9-20 en montre un exemple. Le programme apparaît Figure 9-22, et l'ordinogramme Figure 9-21.

Le programme s'appelle « DELETE », et réside aux adresses 075F à 0799.

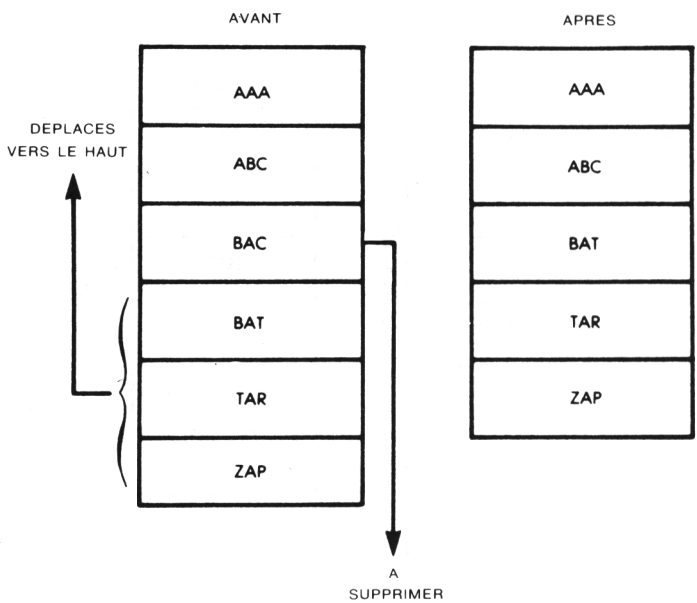


Fig. 9-20 : Suppression de « BAC »

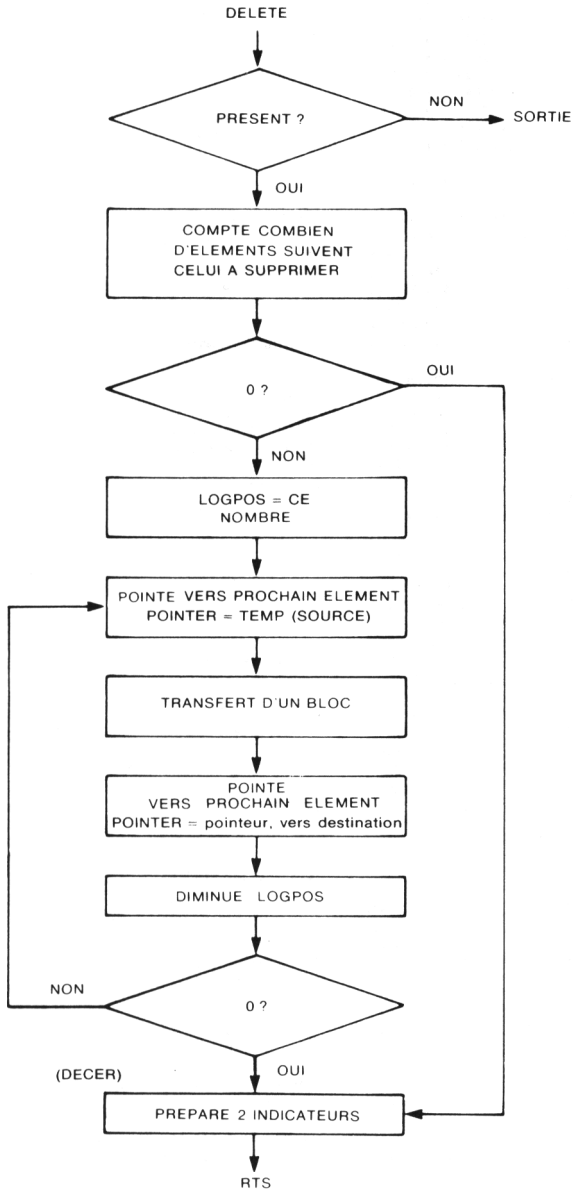


Fig. 9-21 : Ordinogramme de la suppression (table alphabétique)

```
LINE # LOC     CODE       LINE

0002  0000              CLOSE  = $10
0003  0000              CMPRES = $11
0004  0000              TABASE = $12
0005  0000              POINTR = $14
0006  0000              TABLEN = $16
0007  0000              LOGPOS = $17
0008  0000              INCMNT = $18
0009  0000              TEMP   = $19
0010  0000              ENTLEN = $1B
0011  0000              OBJECT = $1C
0012  0000              ;
0013  0000                     * = $600
0014  0600              ;
0015  0600  A9 00       SEARCH LDA #0          ;ZERO FLAGS
0016  0602  85 10              STA CLOSE
0017  0604  85 11              STA CMPRES
0018  0606  A5 12              LDA TABASE      ;INITIALIZE POINTER
0019  0608  85 14              STA POINTR
0020  060A  A5 13              LDA TABASE+1
0021  060C  85 15              STA POINTR+1
0022  060E  A5 16              LDA TABLEN      ;GET TABLE LENGTH
0023  0610  D0 03              BNE DIV
0024  0612  4C E0 06           JMP OUT
0025  0615  4A          DIV    LSR A           ;DIVIDE IT BY 2
0026  0616  69 00              ADC #0          ;ADD BACK IN 1'S BIT
0027  0618  85 17              STA LOGPOS      ;STORE AS LOGICAL POSITION
0028  061A  85 18              STA INCMNT      ;STORE AS INCREMENT VALUE
0029  061C  A6 17              LDX LOGPOS      ;MULTIPLY ENTLEN BY LOGPOS
0030  061E  CA                 DEX             ;..ADDING RESULT TO POINTER
0031  061F  F0 0E              BEQ ENTRY
0032  0621  A5 1B       LOOP   LDA ENTLEN
0033  0623  18                 CLC
0034  0624  65 14              ADC POINTR
0035  0626  85 14              STA POINTR
0036  0628  90 02              BCC LOPP
0037  062A  E6 15              INC POINTR+1
0038  062C  CA           LOPP   DEX
0039  062D  D0 F2              BNE LOOP
0040  062F  A5 18       ENTRY  LDA INCMNT      ;DIVIDE INCREMENT VALUE BY 2
0041  0631  4A                 LSR A
0042  0632  69 00              ADC #0
0043  0634  85 18              STA INCMNT
0044  0636  A0 00              LDY #0          ;COMPARE FIRST LETTERS
0045  0638  B1 1C              LDA (OBJECT),Y
0046  063A  D1 14              CMP (POINTR),Y
0047  063C  D0 11              BNE NOGOOD
0048  063E  C8                 INY             ;COMPARE 2ND LETTERS
0049  063F  B1 1C              LDA (OBJECT),Y
0050  0641  D1 14              CMP (POINTR),Y
0051  0643  D0 0A              BNE NOGOOD
0052  0645  C8                 INY             ;COMPARE 3RD LETTERS
0053  0646  B1 1C              LDA (OBJECT),Y
0054  0648  D1 14              CMP (POINTR),Y
0055  064A  D0 03              BNE NOGOOD
0056  064C  4C E2 06           JMP FOUND
0057  064F  A0 FF       NOGOOD LDY #$FF        ;SET COMPARE RESULT FLAG
0058  0651  90 02              BCC TESTS       ;IF OBJ < POINTR : C=0
0059  0653  A0 01              LDY #1
0060  0655  84 11       TESTS  STY CMPRES
0061  0657  A4 18              LDY INCMNT      ;IS INCR. VALUE @ 1?
0062  0659  88                 DEY
0063  065A  D0 10              BNE NEXT
0064  065C  A5 10              LDA CLOSE       ;CHECK CLOSE FLAG IF IT WAS
0065  065E  F0 08              BEQ MAKCLO      ;IF CLOSE FLAG NOT SET, GO DO IT
0066  0660  38                 SEC
0067  0661  E5 11              SBC CMPRES      ;SEE IF GAVE PASSED WHERE OBJ.
0068  0663  F0 07              BEQ NEXT        ;..SHOULD BE BUT ISN'T
```

Fig. 9-22 : Programmes concernant la liste alphabétique : recherche dichotomique, suppression, insertion

```
0069  0665  4C E0 06          JMP OUT
0070  0668  A5 11       MAKCLO LDA CMPRES      ;SET CLOSE FLAG TO CMPRES
0071  066A  85 10              STA CLOSE
0072  066C  24 11       NEXT   BIT CMPRES
0073  066E  30 35              BMI SUBIT
0074  0670  A5 16              LDA TABLEN       ;SEE IF ADDITIION OF INCMNT
0075  0672  38                 SEC              ;..WILL RUN PAST END OF TABLE
0076  0673  E5 17              SBC LOGPOS
0077  0675  F0 69              BEQ OUT          ;CHECK TO SEE IF AT END OF TABLE ALREADY
0078  0677  E5 18              SBC INCMNT
0079  0679  90 1A              BCC TOOHI
0080  067B  A6 18              LDX INCMNT       ;IS ALL RIGHT, INC POINTER BY
0081  067D  A5 1B       ADDER  LDA ENTLEN       ;..PROPER AMOUNT
0082  067F  18                 CLC
0083  0680  65 14              ADC POINTR
0084  0682  85 14              STA POINTR
0085  0684  90 02              BCC AD1
0086  0686  E6 15              INC POINTR+1
0087  0688  CA          AD1    DEX
0088  0689  D0 F2              BNE ADDER
0089  068B  A5 17              LDA LOGPOS       ;INCREMENT LOGICAL POSITION
0090  068D  18                 CLC
0091  068E  65 18              ADC INCMNT
0092  0690  85 17              STA LOGPOS
0093  0692  4C 2F 06           JMP ENTRY
0094  0695  E6 17       TOOHI  INC LOGPOS       ;INCR. LOGICAL POSITION
0095  0697  A5 1B              LDA ENTLEN       ;MOVE POINTER UP ONE ENTRY
0096  0699  18                 CLC
0097  069A  65 14              ADC POINTR
0098  069C  85 14              STA POINTR
0099  069E  90 35              BCC SETCLO
0100  06A0  E6 15              INC POINTR+1
0101  06A2  4C D5 06           JMP SETCLO
0102  06A5  A5 17       SUBIT  LDA LOGPOS       ;SEE IF INC WILL GO OFF BOTTOM
0103  06A7  38                 SEC              ;.. OF TABLE
0104  06A8  E5 18              SBC INCMNT
0105  06AA  F0 17              BEQ TOOLOW
0106  06AC  90 15              BCC TOOLOW
0107  06AE  85 17              STA LOGPOS       ;SAVE NEW LOGICAL POSITION
0108  06B0  A6 18              LDX INCMNT
0109  06B2  A5 14       SUBLOP LDA POINTR       ;SUBTRACT PROPER AMT. FROM POINTER
0110  06B4  38                 SEC
0111  06B5  E5 1B              SBC ENTLEN
0112  06B7  85 14              STA POINTR
0113  06B9  B0 02              BCS SUB0
0114  06BB  C6 15              DEC POINTR+1
0115  06BD  CA          SUB0   DEX
0116  06BE  D0 F2              BNE SUBLOP
0117  06C0  4C 2F 06           JMP ENTRY
0118  06C3  A6 17       TOOLOW LDX LOGPOS       ;SEE IF POS IS ALREADY 1
0119  06C5  CA                 DEX
0120  06C6  F0 18              BEQ OUT
0121  06C8  C6 17              DEC LOGPOS
0122  06CA  A5 14              LDA POINTR       ;SUB 1 ENTRY FROM POINTER
0123  06CC  38                 SEC
0124  06CD  E5 1B              SBC ENTLEN
0125  06CF  85 14              STA POINTR
0126  06D1  B0 02              BCS SETCLO
0127  06D3  C6 15              DEC POINTR+1
0128  06D5  A9 01       SETCLO LDA #1
0129  06D7  85 18              STA INCMNT
0130  06D9  A5 11              LDA CMPRES
0131  06DB  85 10              STA CLOSE
0132  06DD  4C 2F 06           JMP ENTRY
0133  06E0  A2 FF       OUT    LDX #$FF         ;Z SET IF FOUND
0134  06E2  60          FOUND  RTS
0135  06E3                    ;
0136  06E3                    ;
0137  06E3                    ;
0138  06E3  20 00 06    NEW    JSR SEARCH       ;SEE IF OBJECT IS ALREADY THERE
```

Fig. 9-22 : Programmes concernant la liste alphabétique : recherche dichotomique, suppression, insertion (suite)

```
0139  06E6  F0 76          BEQ OUTE
0140  06E8  A5 16          LDA TABLEN      ;CHECK FOR 0 TABLE
0141  06EA  F0 62          BEQ INSERT
0142  06EC  24 11          BIT CMPRES      ;TEST LAST COMPARE RESULT
0143  06EE  10 05          BPL LOSIDE
0144  06F0  C6 17          DEC LOGPOS      ;SET LOGICAL POSITION SO
0145  06F2  4C 00 07       JMP SETUP       ;..SUB WORKS LATER
0146  06F5  A5 1B    LOSIDE LDA ENTLEN      ;SET POINTER ABOVE WHERE
0147  06F7  18              CLC             ;..OBJECT WILL GO
0148  06F8  65 14          ADC POINTR
0149  06FA  85 14          STA POINTR
0150  06FC  90 02          BCC SETUP
0151  06FE  E6 15          INC POINTR+1
0152  0700  A5 16    SETUP LDA TABLEN      ;SEE HOW MANY ENTRIES THERE
0153  0702  38              SEC             ;..ARE AFTER WHERE OBJ. WILL GO
0154  0703  E5 17          SBC LOGPOS
0155  0705  F0 47          BEQ INSERT
0156  0707  AA              TAX
0157  0708  A8              TAY
0158  0709  88              DEY             ;SEE IF ALREADY POINTING TO
0159  070A  F0 0E          BEQ SETEMP      ;..LAST ENTRY
0160  070C  A5 1B    UPLOOP LDA ENTLEN      ;MOVE POINTER TO LAST ENTRY
0161  070E  18              CLC
0162  070F  65 14          ADC POINTR
0163  0711  85 14          STA POINTR
0164  0713  90 02          BCC SETO
0165  0715  E6 15          INC POINTR+1
0166  0717  88        SETO  DEY
0167  0718  D0 F2          BNE UPLOOP
0168  071A  A5 14    SETEMP LDA POINTR      ;ADD ENTLEN TO POINTER
0169  071C  18              CLC             ;..STORE AT TEMP
0170  071D  65 1B          ADC ENTLEN
0171  071F  85 19          STA TEMP
0172  0721  90 01          BCC SET1
0173  0723  C8              INY             ;IT WAS ALREADY 0
0174  0724  98        SET1  TYA
0175  0725  18              CLC
0176  0726  65 15          ADC POINTR+1
0177  0728  85 1A          STA TEMP+1
0178  072A  A4 1B    MOVER LDY ENTLEN      ;SET Y FOR SHIFT
0179  072C  88      ANOTHR DEY
0180  072D  B1 14          LDA (POINTR),Y  ;MOVE A BYTE
0181  072F  91 19          STA (TEMP),Y
0182  0731  C0 00          CPY #0
0183  0733  D0 F7          BNE ANOTHR
0184  0735  A5 14          LDA POINTR      ;DECR. POINTER AND TEMP
0185  0737  38              SEC             ;..BY ENTLEN
0186  0738  E5 1B          SBC ENTLEN
0187  073A  85 14          STA POINTR
0188  073C  B0 02          BCS M1
0189  073E  C6 15          DEC POINTR+1
0190  0740  CA        M1    DEX
0191  0741  D0 D7          BNE SETEMP
0192  0743  A5 1B          LDA ENTLEN      ;MOVE POINTER BACK TO
0193  0745  18              CLC             ;WHERE OBJ. WILL GO
0194  0746  65 14          ADC POINTR
0195  0748  85 14          STA POINTR
0196  074A  90 02          BCC INSERT
0197  074C  E6 15          INC POINTR+1
0198  074E  A0 00    INSERT LDY #0          ;MOVE OBJECT INTO TABLE
0199  0750  A6 1B          LDX ENTLEN
0200  0752  B1 1C    INNER LDA (OBJECT),Y
0201  0754  91 14          STA (POINTR),Y
0202  0756  C8              INY
0203  0757  CA              DEX
0204  0758  D0 F8          BNE INNER
0205  075A  E6 16          INC TABLEN      ;INCREMENT TABLE LENGTH
0206  075C  A2 FF          LDX #$FF
```

Fig. 9-22 : Programmes concernant la liste alphabétique : recherche dichotomique, suppression, insertion (suite)

```
0207  075E  60          OUTE   RTS             ;Z SET IF NOT DONE
0208  075F                ;
0209  075F                ;
0210  075F                ;
0211  075F  20 00 06     DELETE JSR SEARCH      ;GET ADDR OF OBJECT IN TABLE
0212  0762  D0 35          BNE OUTS            ;SEE IF IT IS THERE
0213  0764  A5 16          LDA TABLEN          ;SEE HOW MANY ENTRIES ARE
0214  0766  38             SEC                 ;..LEFT AFTER OBJ. IN TABLE
0215  0767  E5 17          SBC LOGPOS
0216  0769  F0 2A          BEQ DECER
0217  076B  85 17          STA LOGPOS          ;STORE RESULT AS A COUNTER
0218  076D  A5 1B       BIGLOP LDA ENTLEN      ;SET TEMP & ENTRY ABOVE 1 ENTRY ABOVE OBJ.
0219  076F  18             CLC
0220  0770  65 14          ADC POINTR
0221  0772  85 19          STA TEMP
0222  0774  A9 00          LDA #0
0223  0776  65 15          ADC POINTR+1
0224  0778  85 1A          STA TEMP+1
0225  077A  A6 1B          LDX ENTLEN          ;SET COUNTERS
0226  077C  A0 00          LDY #0
0227  077E  B1 19       BYTE   LDA (TEMP),Y    ;MOVE A BYTE
0228  0780  91 14          STA (POINTR),Y
0229  0782  C8             INY                 ;IS BLOCK MOVED YET?
0230  0783  CA             DEX
0231  0784  D0 F8          BNE BYTE
0232  0786  A5 1B          LDA ENTLEN
0233  0788  18             CLC
0234  0789  65 14          ADC POINTR
0235  078B  85 14          STA POINTR
0236  078D  90 02          BCC D2
0237  078F  E6 15          INC POINTR+1
0238  0791  C6 17       D2     DEC LOGPOS
0239  0793  D0 D8          BNE BIGLOP
0240  0795  C6 16       DECER  DEC TABLEN
0241  0797  A9 00          LDA #0              ;Z SET IF WAS DONE
0242  0799  60          OUTS   RTS
0243  079A                   .END

ERRORS = 0000 <0000>


SYMBOL TABLE

SYMBOL   VALUE

AB1      0688   ADDER    067B   ANOTHR   072C   BIGLOP   076D
BYTE     077E   CLOSE    0010   CMPRES   0011   D2       0791
DECER    0795   DELETE   075F   DIV      0615   ENTLEN   001B
ENTRY    062F   FOUND    06E2   INCMNT   0018   INNER    0752
INSERT   074E   LOGPOS   0017   LOOP     0621   LOPP     062C
LOSIDE   06F5   M1       0740   MAKCLO   0668   MOVER    072A
NEW      06E3   NEXT     066C   NOGOOD   064F   OBJECT   001C
OUT      06E0   OUTE     075E   OUTS     0799   POINTR   0014
SEARCH   0600   SET0     0717   SET1     0724   SETCLO   06D5
SETEMP   071A   SETUP    0700   SUB0     06BD   SUBIT    06A5
SUBLOP   06B2   TABASE   0012   TABLEN   0016   TEMP     001F
TESTS    0655   TOOHI    0695   TOOLOW   06C3   UPLOOP   070C

END OF ASSEMBLY

<
```

Fig. 9-22 : Programmes concernant la liste alphabétique : recherche dichotomique, suppression, insertion (suite)

LISTE CHAINEE

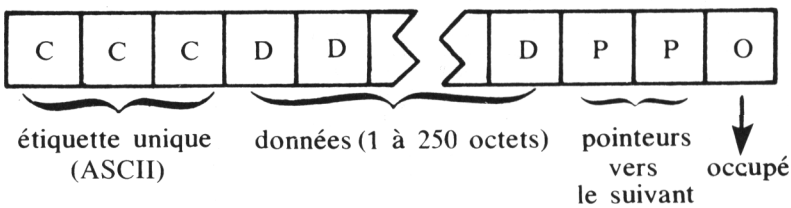
Supposons que la liste chaînée contient, comme précédemment, les trois caractères alphanumériques servant d'étiquette, suivis de 1 à 250 octets de données, d'un pointeur de 2 octets contenant l'adresse de départ du prochain élément, et enfin, d'un marqueur sur un octet. Lorsque ce marqueur contient 1, il empêche la routine d'insertion de substituer un nouvel élément à l'élément considéré.

De plus, un répertoire contient, afin de faciliter les recherches, un pointeur, pointant vers le premier élément correspondant à chaque lettre de l'alphabet. Supposons aussi que, dans le programme, toutes les étiquettes sont des caractères alphabétiques ASCII. Tous les pointeurs, à la fin de la liste, sont mis à la valeur NIL (= pointe vers rien du tout), choisie égale à la base de la table, puisqu'elle ne devrait jamais se trouver dans la liste.

Les programmes d'insertion et de suppression effectuent des opérations évidentes sur les pointeurs. Ils utilisent l'indicateur INDEXD pour préciser si un pointeur vers un objet vient d'un élément précédent de la liste, ou au contraire du répertoire. Les programmes correspondants apparaissent Figure 9-27. La Figure 9-23 montre la structure des données.

Application possible de cette structure de données : un annuaire où chaque personne serait représentée par un code en trois lettres, unique (par exemple des initiales). La zone de données contiendrait l'adresse et le numéro de téléphone (au plus 250 caractères).

Examinons plus en détail la structure Figure 9-23. Le format d'un élément est :



Comme d'habitude, les conventions sont :
ENTLEN : longueur totale d'un élément (en octets).
TABASE : adresse de base de la liste.
TABLEN : nombre d'éléments.

L'adresse de l'objet recherché, OBJECT, est toujours supposée résider dans le registre Y, avant d'entrer dans le programme. Ici, REFBASE pointe vers l'adresse de base du répertoire.

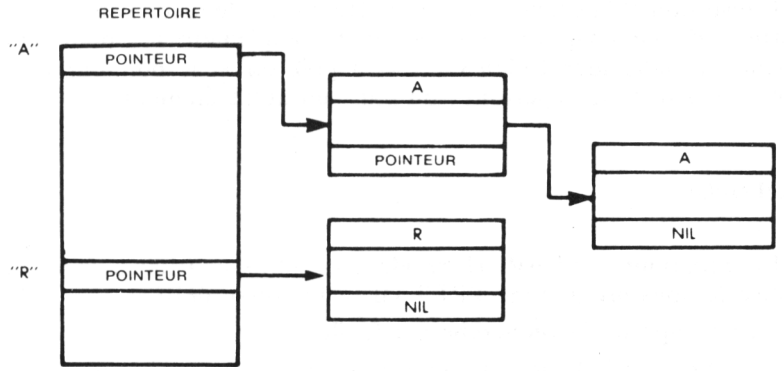


Fig. 9-23

Chaque adresse 16 bits de ce répertoire pointe vers la première occurence dans la liste de la lettre qui lui correspond. Ainsi, chaque groupe d'éléments, dont les étiquettes commencent par la même lettre, forme une liste particulière, à l'intérieur de l'ensemble. Cela ressemble à un carnet d'adresses, et les recherches en sont facilitées. Notez que, lors d'une insertion ou d'une suppression, aucune donnée n'est déplacée. Seuls les pointeurs sont modifiés, comme dans toute liste bien conçue.

Si aucun élément, commençant par une lettre donnée, n'est présent, ou si aucun élément ne suit un élément donné, leurs pointeurs ont la valeur NIL (pointant vers le début de la table). En bas de la table, est rangée une valeur, telle que la différence entre cette valeur et « Z » soit supérieure à « A »-« Z ». Cela constitue une marque de fin de table (EOT). La valeur d'EOT est supposée, ici, occuper la même taille mémoire qu'un élément quelconque. Mais elle pourrait, aussi bien, n'occuper qu'un seul octet.

Les lettres sont supposées alphabétiques, en ASCII. Pour qu'il en soit autrement, il faudrait changer la constante de la routine PRETAB. Le pointeur de la marque de fin de table [EOT] est mis à NIL. Par convention, les pointeurs NIL trouvés en fin de liste ou dans le répertoire, dans le cas d'une lettre sans représentants, pointent vers le début de la table ce qui est une manière de les reconnaître. Nous pourrions utiliser une autre convention. En particulier, une autre marque d'EOT permettrait de réaliser quelques économies de place, car les pointeurs NIL deviendraient inutiles, pour les éléments inexistants.

L'insertion et la suppression sont effectuées, de façon usuelle (cf. partie I de ce chapitre), en modifiant simplement les pointeurs voulus. L'indicateur INDEXD sert à préciser si le pointeur vers l'objet est dans le répertoire, ou dans un autre élément.

Recherche

Le programme SEARCH réside aux adresses 0600 à 0650. Il utilise le sous-programme PRETAB, à l'adresse 06F8.

Le principe de la recherche est facile :

1 — Prendre l'élément du répertoire correspondant à la première lettre de l'étiquette de OBJECT.
2 — Prendre le pointeur trouvé dans le répertoire. Accéder à l'élément. S'il s'agit de NIL, l'élément n'existe pas.
3 — Dans le cas contraire, comparer l'élément à l'objet. S'il y a accord, la recherche a réussi. Sinon, prendre le pointeur vers l'élément suivant.
4 — Recommencer en 2.

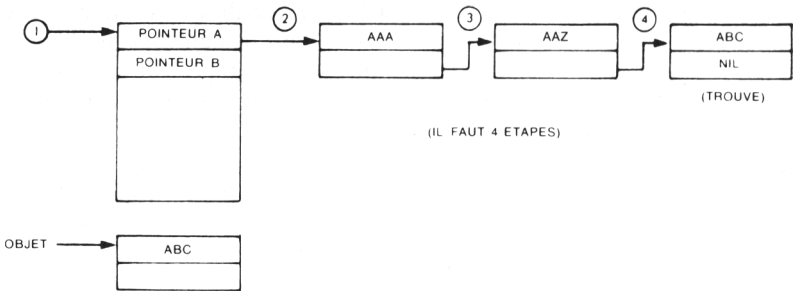


Fig. 9-24 : Recherche dans une liste

Insertion d'un élément

L'insertion est essentiellement une recherche, suivie d'une insertion, après la découverte d'un « NIL ». Un bloc de mémoire est alloué au nouvel élément, au-delà du marqueur d'EOT, en cherchant « NEW ». Il réside aux adresses 0651 à 06BD. La Figure 9-25 en montre un exemple.

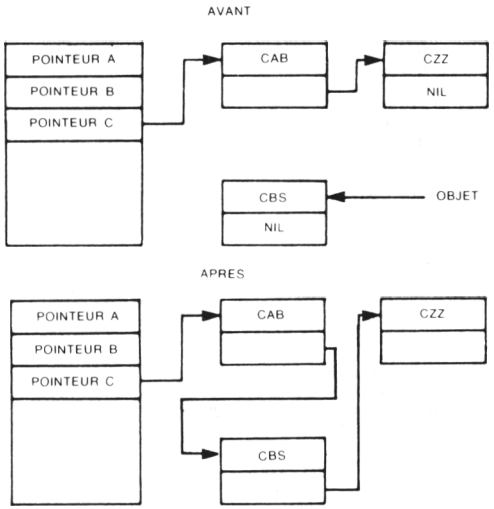


Fig. 9-25 : Exemple d'insertion dans une liste

Suppression d'un élément

L'élément est supprimé en plaçant son marqueur d'occupation à la position « disponible », et en ajustant le pointeur du répertoire, ou de

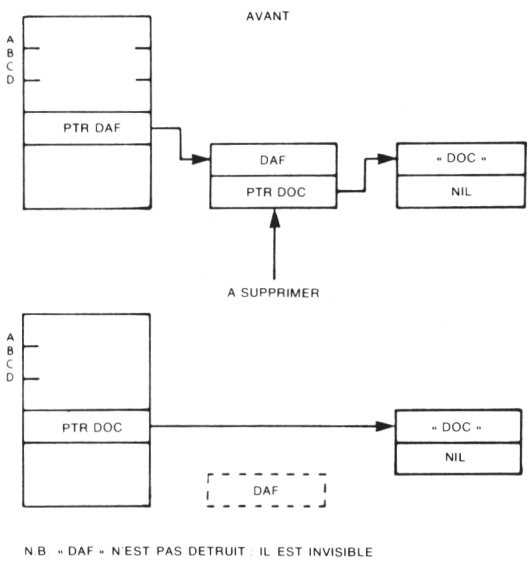


Fig. 9-26 : Suppression dans une liste

```
LINE # LOC     CODE       LINE

0002  0000              INDEXD = $10
0003  0000              INDLOC = $11
0004  0000              POINTR = $13
0005  0000              OBJECT = $15
0006  0000              TEMP   = $17
0007  0000              REFBAS = $19
0008  0000              OLD    = $1B
0009  0000              TABASE = $1D
0010  0000              ENTLEN = $1F
0011  0000              ;
0012  0000                     * = $600
0013  0600              ;
0014  0600  A9 01       SEARCH LDA #1          ;INITIALIZE FLAGS
0015  0602  85 10              STA INDEXD
0016  0604  20 F8 06           JSR PRETAB      ;GET REF. POINTER FOR START
0017  0607  B1 11              LDA (INDLOC),Y  ;PUT IT IN POINTR
0018  0609  85 13              STA POINTR
0019  060B  C8                 INY
0020  060C  B1 11              LDA (INDLOC),Y
0021  060E  85 14              STA POINTR+1
0022  0610  A0 00       ENTRY  LDY #0          ;SEE IF ENTRY IS EOT VALUE
0023  0612  B1 13              LDA (POINTR),Y
0024  0614  C9 7C              CMP #$7C
0025  0616  F0 36              BEQ NOTFND
0026  0618  B1 15              LDA (OBJECT),Y  ;COMPARE FIRST LETTERS
0027  061A  D1 13              CMP (POINTR),Y
0028  061C  90 30              BCC NOTFND
0029  061E  D0 12              BNE NOGOOD
0030  0620  C8                 INY             ;COMPARE SECOND LETTERS
0031  0621  B1 15              LDA (OBJECT),Y
0032  0623  D1 13              CMP (POINTR),Y
0033  0625  90 27              BCC NOTFND
0034  0627  D0 09              BNE NOGOOD
0035  0629  C8                 INY             ;COMPARE THIRD LETTERS
0036  062A  B1 15              LDA (OBJECT),Y
0037  062C  D1 13              CMP (POINTR),Y
0038  062E  90 1E              BCC NOTFND
0039  0630  F0 1E              BEQ FOUND
0040  0632  A5 14       NOGOOD LDA POINTR+1    ;SAVE POINTR FOR POSSIBLE REF.
0041  0634  85 1C              STA OLD+1
0042  0636  A5 13              LDA POINTR
0043  0638  85 1B              STA OLD
0044  063A  A4 1F              LDY ENTLEN      ;GET POINTER FROM ENTRY AND
0045  063C  B1 13              LDA (POINTR),Y  ;..LOAD IT INTO POINTR
0046  063E  AA                 TAX
0047  063F  C8                 INY
0048  0640  B1 13              LDA (POINTR),Y
0049  0642  85 14              STA POINTR+1
0050  0644  8A                 TXA
0051  0645  85 13              STA POINTR
0052  0647  A9 00              LDA #0
0053  0649  85 10              STA INDEXD      ;RESET FLAG
0054  064B  4C 10 06           JMP ENTRY
0055  064E  A9 FF       NOTFND LDA #$FF
0056  0650  60         FOUND  RTS             ;Z SET IF FOUND
0057  0651              ;
0058  0651              ;
0059  0651              ;
0060  0651  20 00 06    NEW    JSR SEARCH      ;SEE IF OBJ. IS ALREADY THERE
0061  0654  F0 67              BEQ OUTE
0062  0656  A5 1D              LDA TABASE      ;LOOK FOR UNOCCUPIED ENTRY
0063  0658  18                 CLC             ;..BLOCK
0064  0659  69 01              ADC #1          ;JUMP PAST EOT VALUE
0065  065B  85 17              STA TEMP
0066  065D  A9 00              LDA #0
0067  065F  65 1E              ADC TABASE+1
0068  0661  85 18              STA TEMP+1
0069  0663  A4 1F              LDY ENTLEN      ;SET Y TO POINT TO OCCUPANCY
```

Fig. 9-27 : Programmes de liste

```
0070  0665  C8              INY             ;..MARKER OF AN ENTRY
0071  0666  C8              INY
0072  0667  A9 01     LOOP   LDA #1          ;TEST FOR OCCUPANCY MARKER
0073  0669  D1 17            CMP (TEMP),Y
0074  066B  D0 16            BNE INSERT
0075  066D  A5 17            LDA TEMP        ;IF IS USED, MOVE TEMP TO NEXT
0076  066F  18              CLC             ;..ENTRY BLOCK
0077  0670  65 1F            ADC ENTLEN
0078  0672  90 02            BCC MORE
0079  0674  E6 18            INC TEMP+1
0080  0676  69 03     MORE   ADC #3
0081  0678  85 17            STA TEMP
0082  067A  A9 00            LDA #0
0083  067C  65 18            ADC TEMP+1
0084  067E  85 18            STA TEMP+1
0085  0680  4C 67 06          JMP LOOP
0086  0683  88         INSERT DEY             ;SET Y BACK TO POINTING TO
0087  0684  88                DEY             ;..TOP OF DATA
0088  0685  88         LOPE   DEY             ;MOVE OBJECT INTO SPACE
0089  0686  B1 15            LDA (OBJECT),Y
0090  0688  91 17            STA (TEMP),Y
0091  068A  C0 00            CPY #0
0092  068C  D0 F7            BNE LOPE
0093  068E  A4 1F            LDY ENTLEN      ;PUT THE VALUE OF POINTR, THE
0094  0690  A5 13            LDA POINTR      ;ENTRY AFTER OBJECT, INTO
0095  0692  91 17            STA (TEMP),Y    ;POINTER AREA OF OBJECT
0096  0694  C8              INY
0097  0695  A5 14            LDA POINTR+1
0098  0697  91 17            STA (TEMP),Y
0099  0699  C8              INY
0100  069A  A9 01            LDA #1          ;SET OCCUPANCY MARKER
0101  069C  91 17            STA (TEMP),Y
0102  069E  A5 10            LDA INDEXD      ;TEST TO SEE IF REF. TABLE
0103  06A0  D0 0D            BNE SETINX      ;..NEEDS READJUSTING
0104  06A2  88              DEY
0105  06A3  A5 18            LDA TEMP+1      ;NO, CHANGE PREVIOUS ENTRY'S
0106  06A5  91 1B            STA (OLD),Y     ;..POINTER
0107  06A7  88              DEY
0108  06A8  A5 17            LDA TEMP
0109  06AA  91 1B            STA (OLD),Y
0110  06AC  4C BB 06          JMP DONE
0111  06AF  20 F8 06  SETINX JSR PRETAB      ;GET ADDRESS OF WHATS TO BE CHANGED
0112  06B2  A5 17            LDA TEMP        ;LOAD ADDR. OF OBJ. THERE
0113  06B4  91 11            STA (INDLOC),Y
0114  06B6  C8              INY
0115  06B7  A5 18            LDA TEMP+1
0116  06B9  91 11            STA (INDLOC),Y
0117  06BB  A9 FF     DONE   LDA #$FF
0118  06BD  60         OUTE   RTS             ;Z CLEAR IF DONE
0119  06BE              ;
0120  06BE              ;
0121  06BE              ;
0122  06BE  20 00 06  DELETE JSR SEARCH       ;GET ADDR OF OBJ.
0123  06C1  D0 34            BNE OUTS
0124  06C3  A4 1F            LDY ENTLEN      ;STORE POINTER AT END
0125  06C5  B1 13            LDA (POINTR),Y  ;..OF OBJECT
0126  06C7  85 17            STA TEMP
0127  06C9  C8              INY
0128  06CA  B1 13            LDA (POINTR),Y
0129  06CC  85 18            STA TEMP+1
0130  06CE  C8              INY
0131  06CF  A9 00            LDA #0          ;CLEAR OCCUPANCY MARKER
0132  06D1  91 13            STA (POINTR),Y
0133  06D3  A5 10            LDA INDEXD      ;SEE IF REF. TABLE NEEDS
0134  06D5  F0 06            BEQ PREINX      ;..READJUSTING
0135  06D7  20 F8 06          JSR PRETAB
0136  06DA  4C EA 06          JMP MOVEIT
0137  06DD  A5 1B     PREINX LDA OLD         ;SET FOR CHANGING PREVIOUS
0138  06DF  18              CLC             ;..ENTRY
```

Fig. 9-27 : Programmes de liste (suite)

```
0139  06E0  65 1F          ADC ENTLEN
0140  06E2  85 11          STA INDLOC
0141  06E4  A9 00          LDA #0
0142  06E6  65 1C          ADC OLD+1
0143  06E8  85 12          STA INDLOC+1
0144  06EA  A5 17    MOVEIT LDA TEMP        ;CHANGE WHAT NEEDS CHANGING
0145  06EC  A0 00          LDY #0
0146  06EE  91 11          STA (INDLOC),Y
0147  06F0  C8             INY
0148  06F1  A5 18          LDA TEMP+1
0149  06F3  91 11          STA (INDLOC),Y
0150  06F5  A9 00          LDA #0
0151  06F7  60       OUTS  RTS              ;Z SET IF DONE
0152  06F8           ;
0153  06F8           ;
0154  06F8           ;
0155  06F8  A0 00    PRETAB LDY #0
0156  06FA  B1 15          LDA (OBJECT),Y
0157  06FC  38             SEC              ;REMOVE ASCII LEADER FROM
0158  06FD  E9 41          SBC #$41         ;..FIRST LETTER IN OBJECT
0159  06FF  0A             ASL A            ;MULTIPLY BY 2
0160  0700  18             CLC
0161  0701  65 19          ADC REFBAS       ;INDEX INTO REF. TABLE
0162  0703  85 11          STA INDLOC
0163  0705  A9 00          LDA #0
0164  0707  65 1A          ADC REFBAS+1
0165  0709  85 12          STA INDLOC+1
0166  070B  60             RTS
0167  070C                 .END

ERRORS = 0000  0000
```

```
SYMBOL TABLE

SYMBOL   VALUE

DELETE   06BE   DONE     06BB   ENTLEN   001F   ENTRY    0610
FOUND    0650   INDEXD   0010   INDLOC   0011   INSERT   0683
LOOP     0667   LOPE     0685   MORE     0676   MOVEIT   06EA
NEW      0651   NOGOOD   0632   NOTFND   064E   OBJECT   0015
OLD      001B   OUTE     06BD   OUTS     06F7   POINTR   0013
PREINX   06DD   PRETAB   06F8   REFBAS   0019   SEARCH   0600
SETINX   06AF   TABASE   001D   TEMP     0017
END OF ASSEMBLY
```

Fig. 9-27 : Programmes de liste (suite)

l'élément précédent. Le programme est nommé « DELETE », et il réside aux adresses 06BE à 06F7. La Figure 9-26 montre un exemple de suppression.

ARBRE BINAIRE

Nous allons maintenant développer des routines typiques de manipulation d'un arbre. Notre structure de base apparaît Figure 9-28.

Il s'agit d'un arbre binaire, dont les nœuds sont des noms de personnes. Les noms seront triés à l'intérieur, en fonction de « clés » qui seront les trois premières lettres de chaque nom. La représentation-mémoire de cette structure d'arbre apparaît Figure 9-29. Nous y distinguons le contenu de chaque nœud, ainsi que les deux liens. Le premier lien, à gauche du nom, est le « rejeton de gauche », l'autre, le « rejeton de droite ». Par exemple, l'élément de la table correspondant à Jones contient deux liens : « 2 » et « 4 ». Le rejeton de gauche est donc l'élément n° 2 (Anderson), et celui de droite le n° 4 (Smith). Un « 0 » signifie « pas de descendance ». La clé d'un rejeton de gauche précède son parent dans l'ordre alphabétique. La clé d'un rejeton de droite suit son parent.

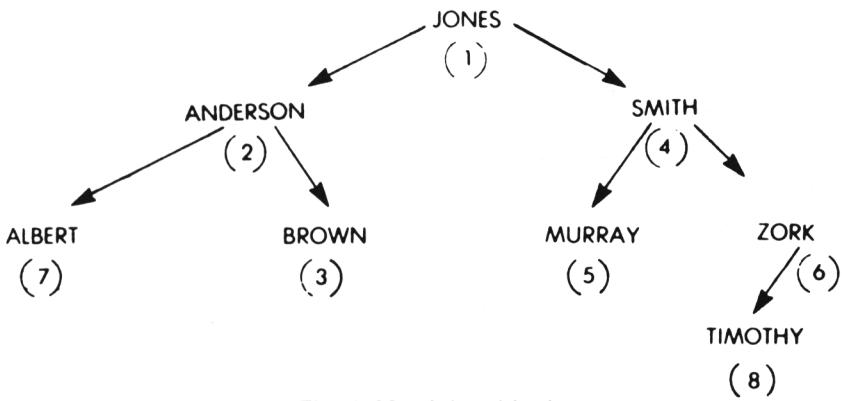


Fig. 9-28 : Arbre binaire

Les deux principaux programmes de gestion d'un arbre sont le programme de « *construction* », et celui de *parcours de l'arbre.* L'élément à insérer est placé dans un tampon. Le programme de construction insère le contenu du tampon dans le nœud approprié.

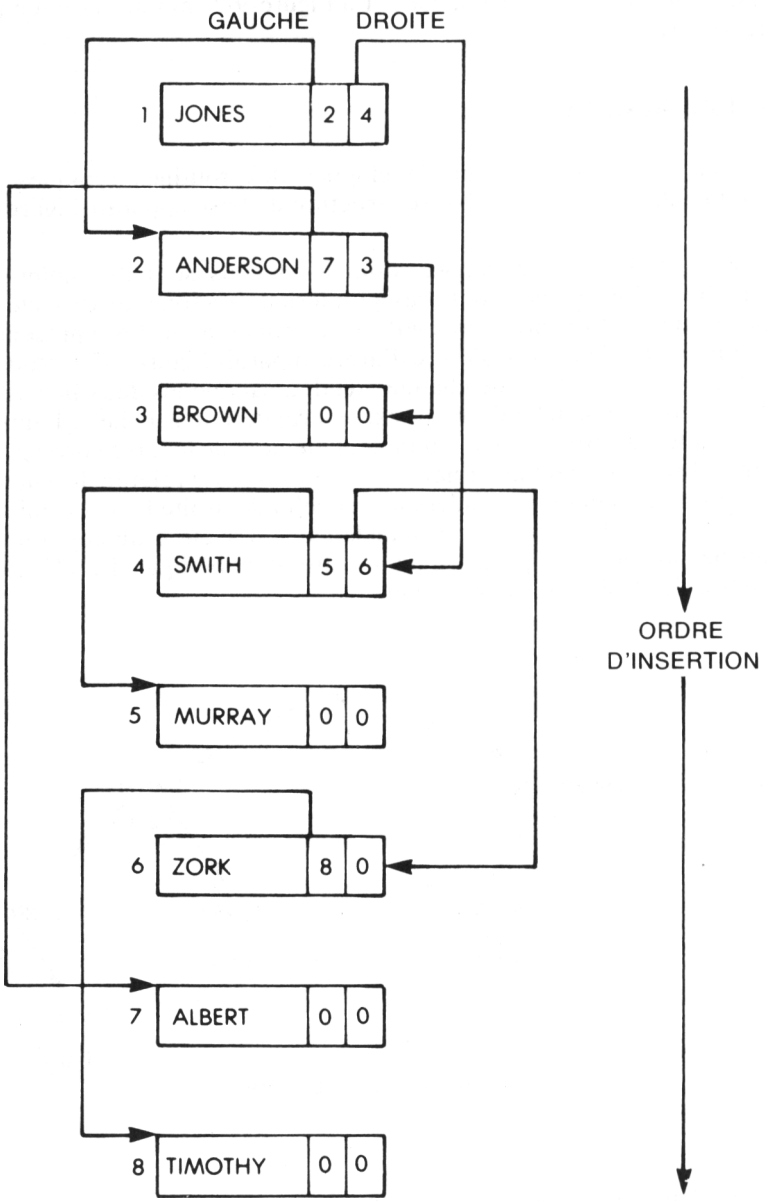


Fig. 9-29 : Représentation de l'arbre en mémoire

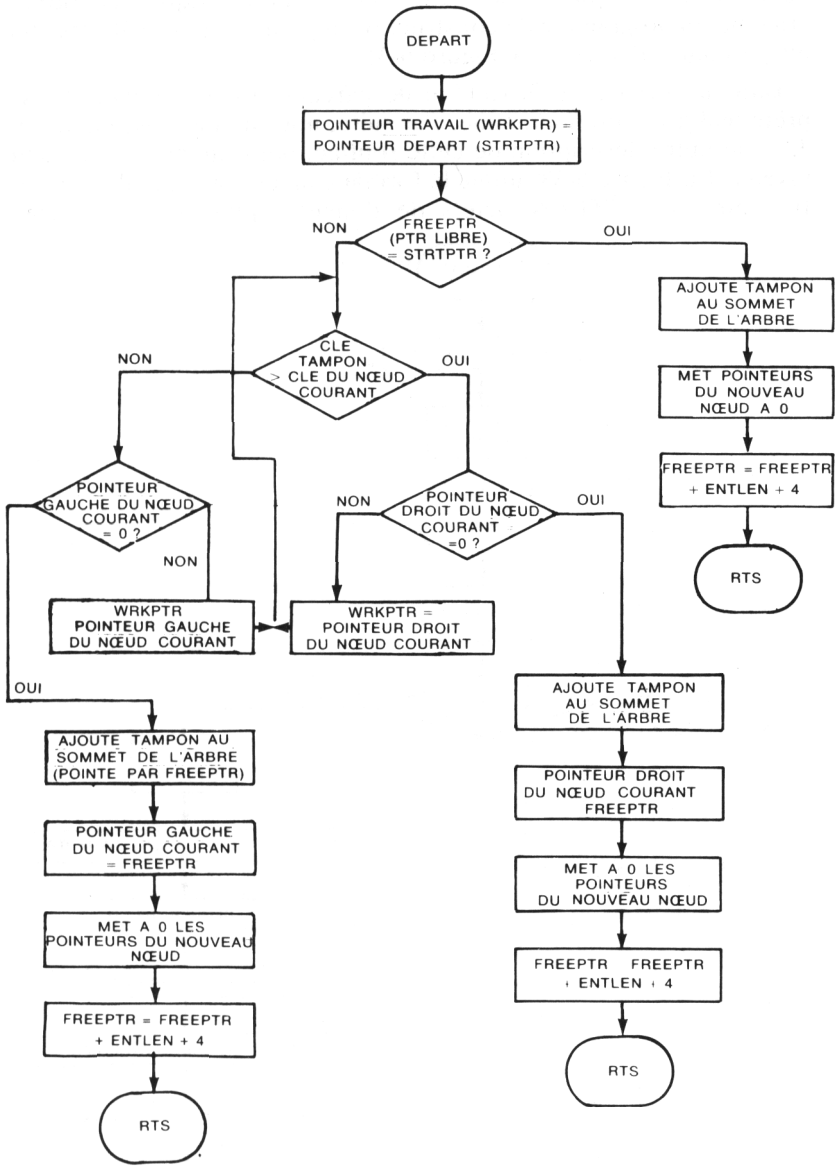


Fig. 9-30 : Ordinogramme du programme de construction d'arbre

Le programme de parcours procède récursivement. Il imprime le contenu de chaque nœud dans l'ordre alphabétique. L'ordinogramme du programme de construction apparaît Figure 9-30, et celui du parcours d'arbre à la Figure 9-31.

Dans la mesure où la routine de parcours est récursive, elle se prête mal à un ordinogramme. Par suite, nous donnerons, Figure 9-32, une autre description de cette routine, dans un langage de haut niveau. La Figure 9-33 donne le format exact d'un nœud de l'arbre. Il comporte ENTLEN octets de données, puis deux pointeurs

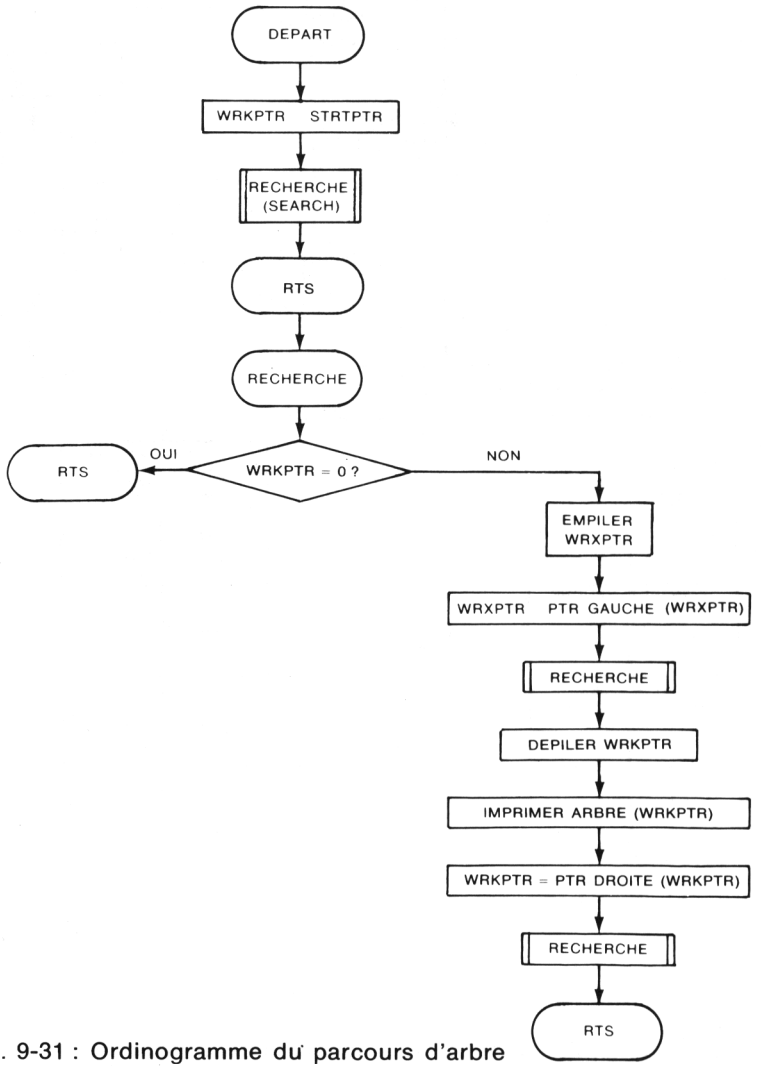


Fig. 9-31 : Ordinogramme du parcours d'arbre

```
PROGRAM parcoursarbre ;
PROCEDURE recherche ;
BEGIN
IF pointtravail # 0 THEN BEGIN
                         empiler (ptrtravail)
                         ptrtravail : = ptrgauche [ptrtravail]
                         recherche ;
                         depiler (ptrtravail) ;
                         writeln (tree [ptrtravail]) ;
                         ptrtravail : = ptrdroite [ptrtravail]
                         recherche
                         END
END ;
BEGIN (parcoursarbre)
ptrtravail : = ptrdépart ;
recherche
END.
```

Fig. 9-32 : Algorithme de parcours d'arbre

16 bits (le pointeur droite et le pointeur gauche). Pour éviter les confusions, notez que la représentation de la Figure 9-29 a été simplifiée, et que le pointeur de droite apparaît, en mémoire, à gauche du pointeur de gauche.

La Figure 9-34 montre les allocations-mémoire utilisées dans le programme, qui apparaît lui-même Figure 9-37.

La routine INSERT réside aux adresses 0200 à 0289. La clé de l'objet à insérer va être comparée à l'élément. Si elle lui est supérieure, on se déplacera sur la droite. Si elle lui est inférieure, sur la gauche. Le processus sera répété jusqu'à la découverte d'un lien nul, ou d'un encadrement correct pour le nouveau nœud (autrement dit un nœud plus grand et un plus petit). Le nouveau nœud sera alors inséré, et les liens appropriés établis.

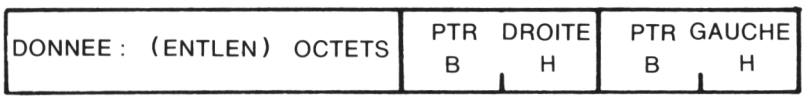


Fig. 9-33 : Nœuds d'un arbre

La routine de parcours (TRAVERSE) réside aux adresses 0285 à 02D6, et les utilitaires OUT, ADD et CLRPTR aux adresses 0207 à 02FE (cf. Figure 9-37).

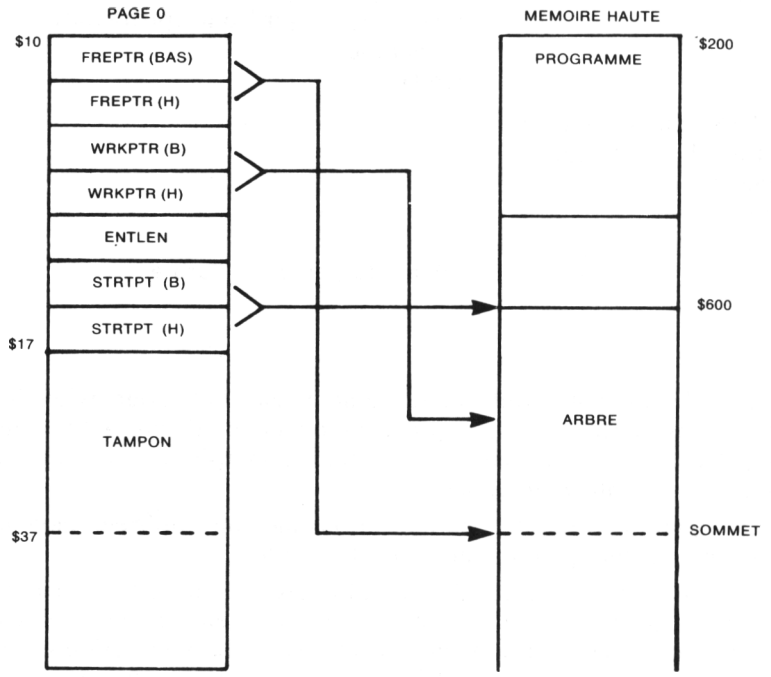


Fig. 9-34 : Carte d'implantation mémoire pour les arbres

La Figure 9-35 montre un exemple d'insertion dans l'arbre, et la Figure 9-36 un exemple de parcours.

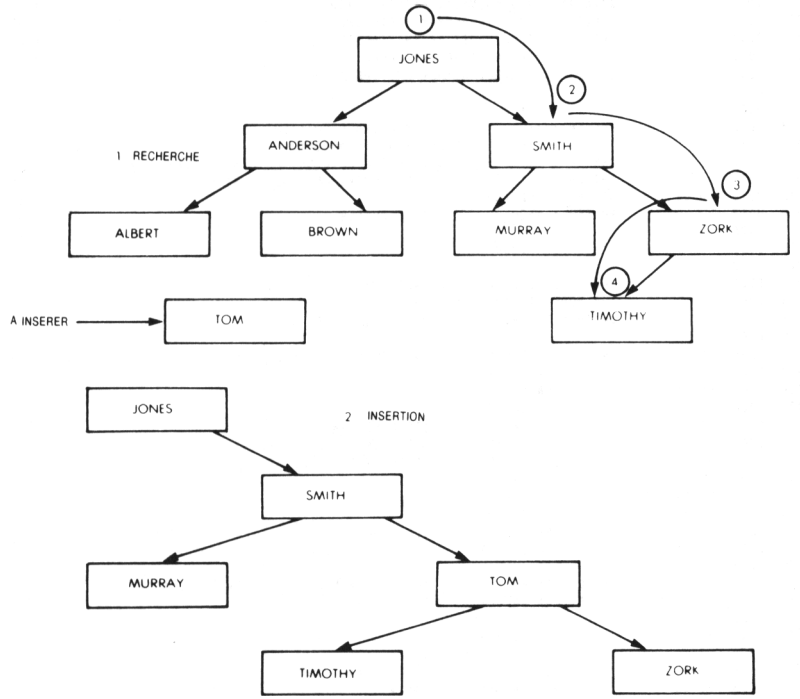


Fig. 9-35 : Insertion d'un élément dans un arbre

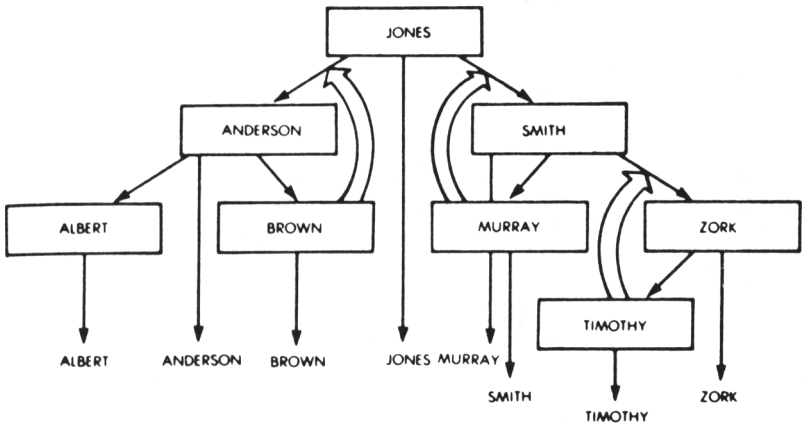


Fig. 9-36 : Parcours d'un arbre (listing)

```
0002  0000                ;TREE MANAGEMENT PROGRAM.
0003  0000                ;2 ROUTINES: ONE, WHEN CALLED, PLACES
0004  0000                ;THE CONTENTS OF THE BUFFER INTO THE
0005  0000                ;TREE; AND THE SECOND TRAVERSES
0006  0000                ;THE TREE RECURSIVELY, PRINTING ITS
0007  0000                ;NODE CONTENTS IN ALPHANUMERIC ORDER.
0008  0000                ;NOTE: 'ENTLEN' MUST BE INITIALIZED
0009  0000                ;AND 'FREPTR' MUST BE SET EQUAL TO
0010  0000                ;'STRTPTR' BEFORE EITHER ROUTINE IS USED.
0011  0000                ;
0012  0000                      * = $10
0013  0010                FREPTR *=*+2          ;FREE SPACE POINTER: POINTS TO
0014  0012                         ;NEXT FREE LOCATION IN MEMORY.
0015  0012                WRKPTR *=*+2          ;WORKING POINTER, POINTS TO CURRENT NODE.
0016  0014                ENTLEN *=*+1          ;TREE ENTRY LENGTH, IN BYTES.
0017  0015  00 06         STRTPT .WORD $600
0018  0017                BUFFER *=*+20         ;I/O BUFFER.
0019  002B                ;
0020  002B                      * = $200
0021  0200                ;
0022  0200                ;ROUTINE TO BUILD TREE: ADDS ONE DATA UNIT,
0023  0200                ;OR NODE, TO TREE. MUST BE CALLED
0024  0200                ;WITH DATA UNIT TO BE ADDED IN 'BUFFER'.
0025  0200                ;
0026  0200  A5 15         INSERT LDA STRTPT     ;WORKPOINTER <= FREEPOINTER.
0027  0202  85 12                STA WRKPTR
0028  0204  A5 16                LDA STRTPT+1
0029  0206  85 13                STA WRKPTR+1
0030  0208  A5 10                LDA FREPTR      ;IF FREEPOINTER =
0031  020A  C5 15                CMP STRTPT      ;STARTING LOCATION POINTER,
0032  020C  D0 0D                BNE INLOOP      ;GOTO INSERTION LOOP.
0033  020E  A5 11                LDA FREPTR+1
0034  0210  C5 16                CMP STRTPT+1
0035  0212  D0 07                BNE INLOOP
0036  0214  20 D7 02             JSR ADD         ;LOAD BUFFER INTO CURRENT POSITION.
0037  0217  20 E4 02             JSR CLRPTR      ;SET POINTERS OF CURRENT NODE TO 0.
0038  021A  60                   RTS             ;DONE ADDING 1ST NODE.
0039  021B  A0 00         INLOOP LDY #0          ;COMPARE BUFFER TAG TO TAG OF CURRENT
0040  021D  B9 17 00     CMPLP  LDA BUFFER,Y    ;LOCATION...
0041  0220  D1 12                CMP (WRKPTR),Y
0042  0222  90 33                BCC LESSTN      ;BUFR TAG LOWER: ADD BUFFER TO
0043  0224                            ;LEFT SIDE OF TREE.
0044  0224  F0 02                BEQ NXT         ;TAGS EQUAL, TRY NEXT CHR. IN TAGS.
0045  0226  B0 05                BCS GRTNEQ      ;BUFR TAG GREATER, ADD BUFR TO
0046  0228                          ;RIGHT SIDE OF TREE.
0047  0228  C8                NXT    INY
0048  0229  C9 04                   CMP #4          ;3 CHRS. COMPARED?
0049  022B  D0 F0                   BNE CMPLP       ;NO, CHECK NEXT CHR.
0050  022D  A4 14         GRTNEQ LDY ENTLEN      ;DOES
0051  022F  B1 12                LDA (WRKPTR),Y  ;RIGHT POINTER OF CURRENT NODE = 0 ?
0052  0231  D0 15                BNE NXRNOD      ;IF NOT, MOVE DOWN/RIGHT IN TREE.
0053  0233  C8                   INY
0054  0234  B1 12                LDA (WRKPTR),Y
0055  0236  D0 10                BNE NXRNOD
0056  0238  A5 11                LDA FREPTR+1    ;SET RIGHT POINTER OF CURRENT
0057  023A  91 12                STA (WRKPTR),Y  ;NODE = FREEPOINTER.
0058  023C  88                   DEY
0059  023D  A5 10                LDA FREPTR
0060  023F  91 12                STA (WRKPTR),Y
0061  0241  20 D7 02             JSR ADD         ;ADD BUFFER TO TREE.
0062  0244  20 E4 02             JSR CLRPTR      ;CLEAR POINTERS OF NEW NODE.
0063  0247  60                   RTS             ;DONE, NEW RIGHT NODE ADDED.
0064  0248  A4 14         NXRNOD LDY ENTLEN      ;SET WORKING POINTER
0065  024A  B1 12                LDA (WRKPTR),Y; RIGHT POINTER OF CURRENT NODE.
0066  024C  AA                   TAX
0067  024D  C8                   INY
0068  024E  B1 12                LDA (WRKPTR),Y
0069  0250  85 13                STA WRKPTR+1
0070  0252  86 12                STX WRKPTR
0071  0254  4C 1B 02             JMP INLOOP      ;TRY NEW CURRENT NODE.
```

Fig. 9-37 : Programmes sur les arbres

```
0072  0257  A4 14     LESSTN LDY ENTLEN      ;DOES LEFT POINTER OF
0073  0259  C8                INY             ;CURRENT NODE = 0 ?
0074  025A  C8                INY
0075  025B  B1 12              LDA (WRKPTR),Y
0076  025D  D0 15              BNE NXLNOD      ;IF SO, MOVE DOWN/LEFT IN TREE.
0077  025F  C8                INY
0078  0260  B1 12              LDA (WRKPTR),Y
0079  0262  D0 10              BNE NXLNOD
0080  0264  A5 11              LDA FREPTR+1    ;SET LEFT POINTER OF CURRENT NODE TO
0081  0266  91 12              STA (WRKPTR),Y  ;POINT TO NEW NODE.
0082  0268  88                DEY
0083  0269  A5 10              LDA FREPTR
0084  026B  91 12              STA (WRKPTR),Y
0085  026D  20 B7 02           JSR ADD         ;ADD NEW NODE CONTENTS.
0086  0270  20 E4 02           JSR CLRPTR      ;CLEAR POINTERS OF NEW NODE.
0087  0273  60                RTS             ;DONE, NEW LEFT NODE ADDED.
0088  0274  A4 14     NXLNOD LDY ENTLEN      ;SET WORKING POINTER =
0089  0276  C8                INY             ;LEFT POINTER OF CURRENT NODE.
0090  0277  C8                INY
0091  0278  B1 12              LDA (WRKPTR),Y
0092  027A  AA                TAX
0093  027B  C8                INY
0094  027C  B1 12              LDA (WRKPTR),Y
0095  027E  85 13              STA WRKPTR+1
0096  0280  86 12              STX WRKPTR
0097  0282  4C 1B 02           JMP INLOOP      ;TRY NEW CURRENT NODE.
0098  0285              ;
0099  0285              ;TREE TRAVERSER : LISTS NODES OF TREE
0100  0285              ;IN ALPHANUMERICAL ORDER.
0101  0285              ;OUTPUT ROUTINE TO XFER BUFFER TO OUTPUT
0102  0285              ;DEVICE IS NEEDED.
0103  0285              ;
0104  0285  A5 15     TRVRSE LDA STRTPT      ;WORKING POINTER <= START POINTER.
0105  0287  85 12              STA WRKPTR
0106  0289  A5 16              LDA STRTPT+1
0107  028B  85 13              STA WRKPTR+1
0108  028D  A5 13     SEARCH LDA WRKPTR+1
0109  028F  A6 12              LDX WRKPTR      ;IF WORKING POINTER <> 0,
0110  0291  D0 07              BNE OK          ;CONTINUE;
0111  0293  A4 13              LDY WRKPTR+1
0112  0295  D0 03              BNE OK
0113  0297  4C C6 02           JMP RETN        ;ELSE, RETURN.
0114  029A  48        OK      PHA             ;PUSH WORKING POINTER
0115  029B  8A                TXA             ;ONTO STACK.
0116  029C  48                PHA
0117  029D  A4 14              LDY ENTLEN      ;SET WORKING POINTER =
0118  029F  C8                INY             ;LEFT POINTER OF CURRENT NODE.
0119  02A0  C8                INY
0120  02A1  B1 12              LDA (WRKPTR),Y
0121  02A3  AA                TAX
0122  02A4  C8                INY
0123  02A5  B1 12              LDA (WRKPTR),Y
0124  02A7  85 13              STA WRKPTR+1
0125  02A9  86 12              STX WRKPTR
0126  02AB  20 8D 02           JSR SEARCH      ;SEARCH NEW NODE, RECURSIVELY.
0127  02AE  68                PLA             ;POP OLD CURRENT NODE INTO WORKING POINTER.
0128  02AF  85 12              STA WRKPTR
0129  02B1  68                PLA
0130  02B2  85 13              STA WRKPTR+1
0131  02B4  20 C7 02           JSR OUT         ;OUTPUT CURRENT NODE CONTENTS.
0132  02B7  A4 14              LDY ENTLEN      ;SET WORKING POINTER =
0133  02B9  B1 12              LDA (WRKPTR),Y  ;CURRENT NODE'S RIGHT POINTER.
0134  02BB  AA                TAX
0135  02BC  C8                INY
0136  02BD  B1 12              LDA (WRKPTR),Y
0137  02BF  85 13              STA WRKPTR+1
0138  02C1  86 12              STX WRKPTR
0139  02C3  20 8D 02           JSR SEARCH      ;SEARCH NEW NODE.
0140  02C6  60         RETN    RTS             ;DONE, RETURN.
```

Fig. 9-37 : Programmes sur les arbres (suite)

```
0141  02C7                    ;
0142  02C7                    ;BUFFER OUTPUT ROUTINE.
0143  02C7                    ;
0144  02C7  A0 00       OUT   LDY #0
0145  02C9  B1 12       XFR   LDA (WRKPTR),Y  ;GET CHR. FROM CURRENT NODE.
0146  02CB  99 17 00           STA BUFFER,Y    ;PUT IN BUFFER.
0147  02CE  C8                 INY             ;REPEAT UNTIL...
0148  02CF  C4 14              CPY ENTLEN      ;ALL CHARACTERS XFERRED.
0149  02D1  D0 F6              BNE XFR
0150  02D3  EA                 NOP             ;INSERT CALL TO SUBROUTINE
0151  02D4  EA                 NOP             ;WHICH OUTPUTS BUFFER HERE.
0152  02D5  EA                 NOP
0153  02D6  60                 RTS             ;DONE.
0154  02D7                    ;
0155  02D7                    ;ROUTINE WHICH PLACES BUFFER
0156  02D7                    ;CONTENTS IN NEW NODE.
0157  02D7                    ;
0158  02D7  A0 00       ADD   LDY #0
0159  02D9  B9 17 00    MOV   LDA BUFFER,Y    ;GET CHR. FROM BUFFER.
0160  02DC  91 10              STA (FREPTR),Y  ;STORE IN NEW NODE.
0161  02DE  C8                 INY             ;REPEAT UNTIL...
0162  02DF  C4 14              CPY ENTLEN      ;ALL CHRS XFERRED.
0163  02E1  D0 F6              BNE MOV
0164  02E3  60                 RTS             ;DONE.
0165  02E4                    ;
0166  02E4                    ;ROUTINE TO CLEAR POINTERS OF NEW NODE,
0167  02E4                    ;AND UPDATE FREE SPACE POINTER.
0168  02E4                    ;
0169  02E4  A4 14       CLRPTR LDY ENTLEN     ;SET UP INDEX TO POINT
0170  02E6                                  ;TO TOP OF POINTER LOCATIONS.
0171  02E6  A9 00              LDA #0
0172  02E8  A2 04              LDX #4          ;LOOP 4X TO CLEAR POINTERS
0173  02EA  91 10       CLRLP  STA (FREPTR),Y  ;CLEAR POINTER LOCATION.
0174  02EC  C8                 INY             ;POINT TO NEXT POINTER LOCATION.
0175  02ED  CA                 DEX
0176  02EE  D0 FA              BNE CLRLP       ;LOOP IF NOT DONE.
0177  02F0  A5 14              LDA ENTLEN      ;GET ENTRY LENGTH,
0178  02F2  18                 CLC             ;AND ADD 4 FOR POINTER SPACE.
0179  02F3  69 04              ADC #4
0180  02F5  65 10              ADC FREPTR      ;ADD TO FREE SPACE POINTER TO
0181  02F7  90 02              BCC CC          ;UPDATE IT.
0182  02F9  E6 11              INC FREPTR+1    ;TAKE CARE OF OVERFLOWS.
0183  02FB  85 10       CC     STA FREPTR      ;RESTORE UPDATED FREE SPACE PTR.
0184  02FD  60                 RTS             ;DONE.
0185  02FE                     .END

ERRORS = 0000 <0000
END OF ASSEMBLY
```

Fig. 9-37 : Programmes sur les arbres (suite)

Note sur les arbres

Les arbres binaires peuvent être construits et parcourus de façons différentes. Par exemple, notre arbre pourrait être représenté, autrement, de la façon suivante :

Il faudrait alors le parcourir en ordre :

1 — lister la racine,
2 — parcourir le sous-arbre gauche,
3 — parcourir le sous-arbre droit.

Il existe beaucoup d'autres techniques et conventions.

ALGORITHME DE HACHAGE

Comment placer les identificateurs, de façon systématique, dans un espace mémoire limité de telle sorte qu'ils puissent être retrouvés facilement ? Il s'agit là d'un problème classique, lors de la création de structures de données. Malheureusement, sauf s'il s'agit de nombres entiers consécutifs, les identificateurs supportent mal d'être placés, sans lacunes, en mémoire. Les noms sont souvent placés en mémoire par ordre alphabétique, ce qui permet de les retrouver facilement. Cette opération est, ici, inacceptable, dans la mesure où elle nous contraindrait à recourir à une énorme mémoire. Il faudrait, notamment, allouer un bloc-mémoire à chaque nom possible. La méthode dite de hachage, qui attribue à chaque nom à entrer un nombre unique (ou presque) peut convenir à la résolution de ce problème. La fonction mathématique utilisée pour effectuer le hachage doit être simple, de sorte que l'algorithme soit rapide, mais cependant assez perfectionnée pour que la distribution des noms possibles sur l'espace-mémoire disponible soit aléatoire. Le nombre résultant peut alors être utilisé comme index vers l'emplacement mémoire effectif. La recherche est ainsi très rapide, et c'est la raison pour laquelle le hachage est utilisé pour les répertoires de noms alphabétiques.

Aucun algorithme ne peut garantir que deux noms ne vont pas aboutir au même emplacement-mémoire (c'est une « collision »). Il importe donc de développer une technique capable de résoudre ce problème. Un bon algorithme de hachage répartit les noms uniformément sur l'espace-mémoire disponible, et il autorise une récupération efficace des valeurs, une fois ces dernières stockées dans une table. L'algorithme de hachage utilisé ici est très simple : il suffit d'effectuer le ou exclusif de tous les octets de la clé. Après chaque addition, une rotation est opérée pour rendre la répartition plus aléatoire.

Pour résoudre le problème des collisions, on aura recours à la technique séquentielle simple, dite « technique d'adressage ouvert séquentiel ». Elle consiste à allouer à l'élément le prochain bloc séquentiellement disponible dans la table. Cela ressemble à un carnet d'adresses. Supposons que nous souhaitons ajouter le nom « SMITH », mais que la page des S soit pleine. Nous le mettrons à la page suivante (page des T). Notons qu'une nouvelle collision (avec un nom commençant par T) n'est pas inévitable. Le nom commençant par S peut être supprimé avant l'entrée d'un nom commençant par T.

On peut également envisager une chaîne de collisions. Si la chaîne est longue, alors que la table n'est pas pleine, c'est que l'algorithme de hachage est mauvais.

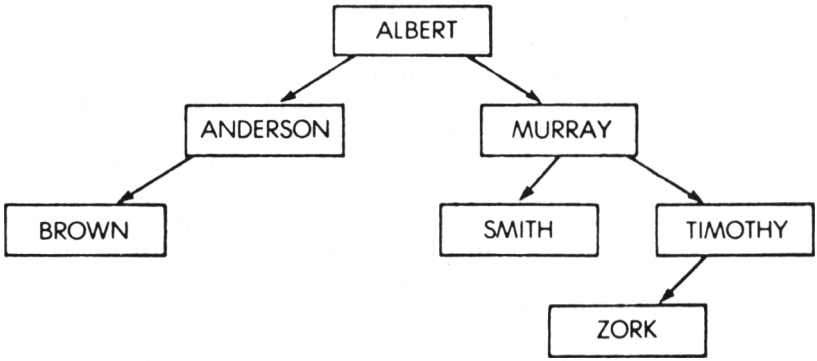


Fig. 9-38 : Arbre préordonné

Comme il est commode d'utiliser une puissance de deux pour le format des données, leur longueur sera 8 caractères : six pour la clé, et deux pour la donnée. Une telle situation courante lors, par exemple, de la création de la table des symboles, pour un assembleur. Jusqu'à 6 caractères peuvent être alloués au symbole, et 2 octets à l'adresse qu'il représente.

Le temps requis par la recherche d'éléments dans la table de hachage, ne dépend pas de la table, mais de son degré de remplissage. Généralement, si le taux d'occupation est maintenu en-dessous de 80 %, le temps d'accès sera rapide (un ou deux essais). Il appartient à la routine d'appel de garder trace du taux d'occupation, et d'éviter tout débordement.

La Figure 9-39 montre l'accroissement du taux d'accès en fonction du degré de remplissage. Les principales routines du programme

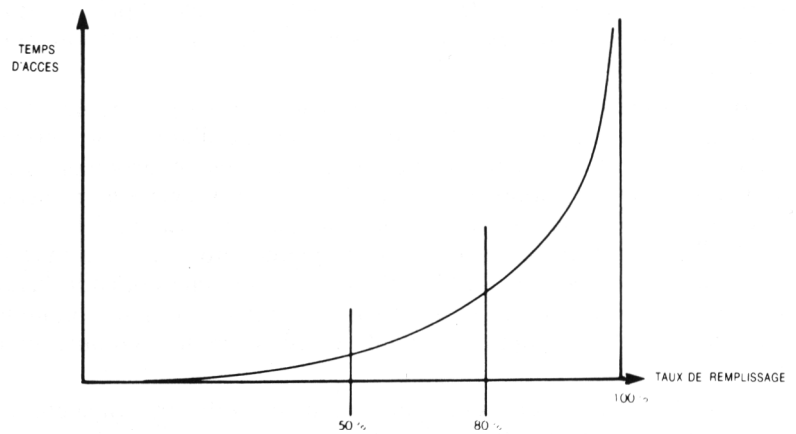


Fig. 9-39 : Temps d'accès en fonction du degré de remplissage

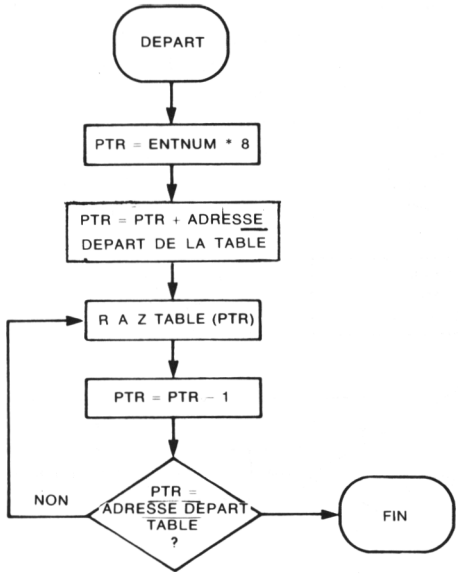


Fig. 9-40 : Sous-programme d'initialisation

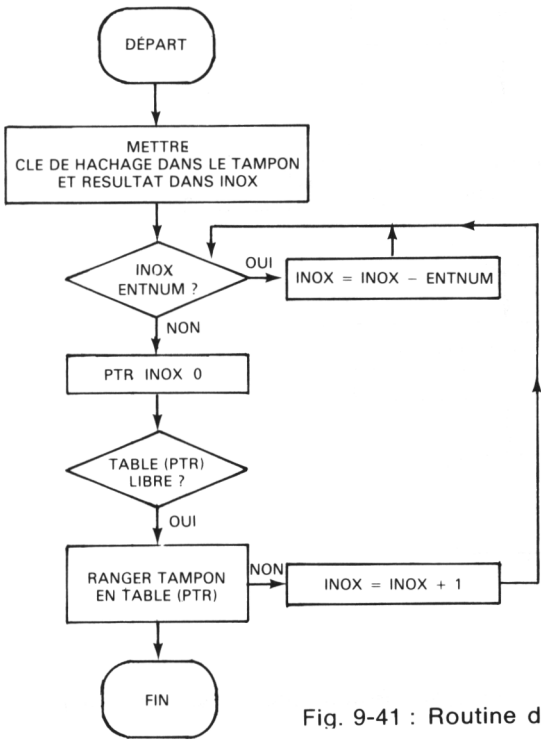


Fig. 9-41 : Routine de stockage

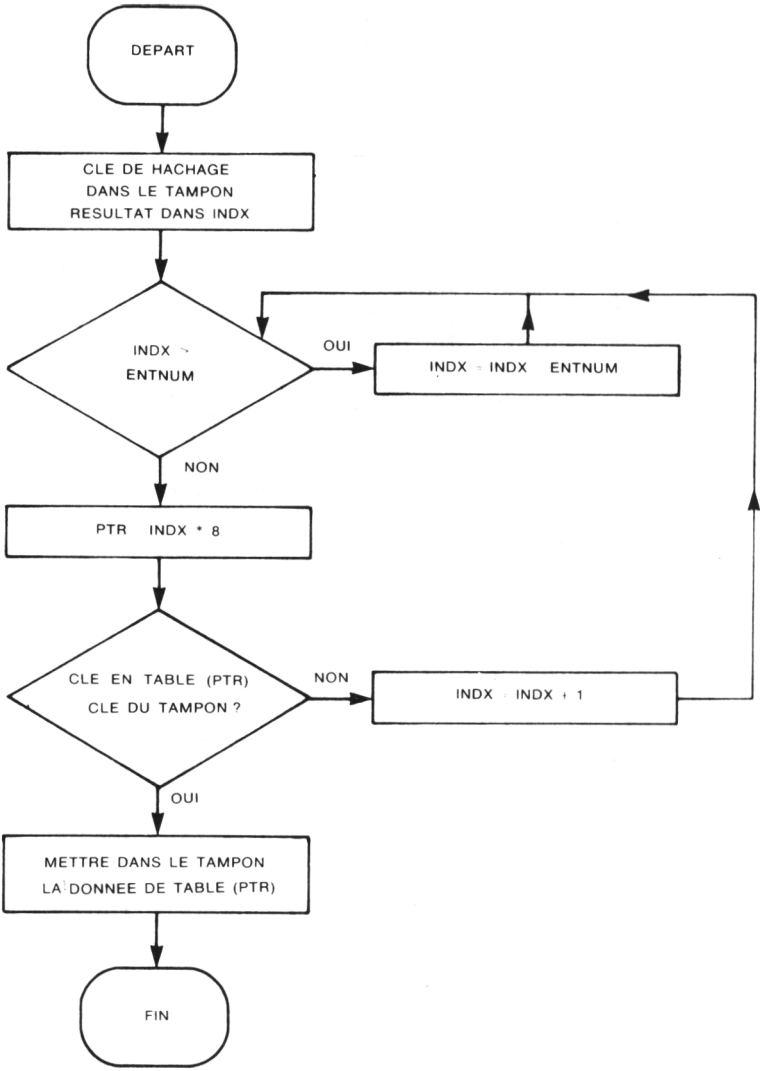


Fig. 9-42 : Routine de recherche (FND)

sont l'initialisation INIT (Figure 9-40), la routine de stockage (Figure 9-41), la routine de recherche (Figure 9-42) et la routine de hachage (Figure 9-43). L'implantation mémoire est montrée Figure 9-44 et le programme, Figure 9-45. Le programme a pour objectif de démontrer les principaux algorithmes utilisés dans un véritable mécanisme de hachage. Pour incorporer ces programmes dans un programme effectif, il est vivement conseillé d'ajouter les fonctions habituelles de gestion, afin d'éviter les situations imprévues.

En particulier, il faut se prémunir contre l'éventualité d'une table pleine ou d'une clé incorrecte, qui pourraient entraîner un bouclage indéfini du programme. Le lecteur est vivement encouragé à étudier ce programme. Non seulement parce qu'il démystifie un algorithme de hachage, mais encore parce qu'il résout un problème pratique important, que l'on rencontre au moment de l'écriture d'un assembleur, ou de toute autre structure où il s'agit de gérer, de façon efficace, des tables de noms avec valeurs correspondantes.

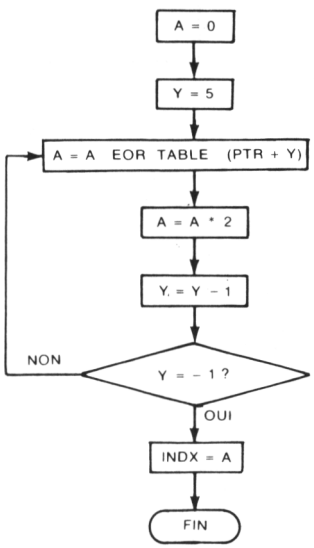


Fig. 9-43 : Routine de hachage

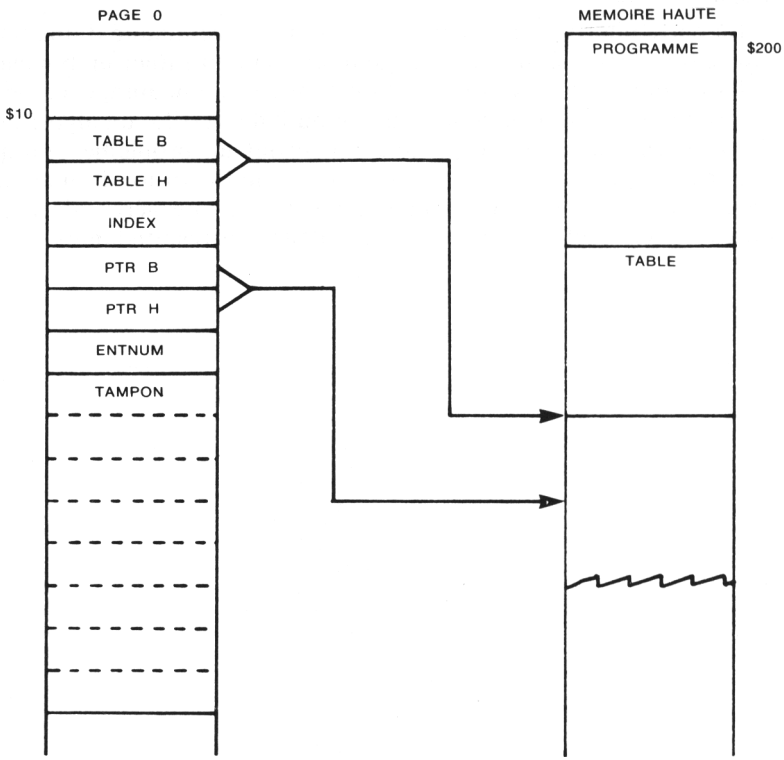


Fig. 9-44 : Implantation mémoire pour le hachage

```
LINE # LOC     CODE       LINE

0002  0000              ;PROGRAM TO STORE ASSEMBLER SYMBOLS IN A
0003  0000              ;TABLE, ACCESSED BY HASHING. THE SYMBOLS
0004  0000              ;ARE 6 CHRS, DATA 2. THE MAXIMUM NUMBER OF
0005  0000              ;8-BYTE UNITS TO BE STORED IN THE TABLE
0006  0000              ;SHOULD BE IN 'ENTNUM', BEGINNING ADDRESS OF
0007  0000              ;TABLE SHOULD BE IN 'TABLE'. NOTE THAT
0008  0000              ;TABLE MUST BE INITIALIZED WITH ROUTINE
0009  0000              ; 'INIT' PRIOR TO USE.
0010  0000              ;IT IS THE RESPONSIBILITY OF THE CALLING
0011  0000              ;PROGRAM NO TO EXCEED THE TABLE SIZE.
0012  0000              ;
0013  0000                     * = $10
0014  0010  00 06       TABLE  .WORD $600       ;STARTING ADDRESS OF TABLE.
0015  0012              INDX   *=*+1            ;NUMBER OF DATA UNIT TO BE ACCESSED.
0016  0013              PTR    *=*+2            ;POINTER TO DATA UNIT IN TABLE.
0017  0015              ENTNUM *=*+1            ;NUMBER OF ENTRIES IN TABLE (256 MAX)
0018  0016              BUFFER *=*^8            ;INPUT/ OUTPUT BUFFER.
0019  001E              ;
0020  001E                     * = $200
0021  0200              ;
0022  0200              ;ROUTINE  INIT' : INITIALIZES TABLE
0023  0200              ;TO ZEROES.
0024  0200              ;
0025  0200 A5 15       INIT   LDA ENTNUM
0026  0202  85 13              STA PTR          ;STORE 0 OF ENTRIES IN POINTER
0027  0204  20 72 02           JSR SHADD        ;MULTIPLY PTR*8, ADD TABLE POINTER.
0028  0207  A2 00              LDX #0           ;CLEAR X FOR INDIRECT ADDRESSING.
0029  0209  A9 00       CLRLP  LDA #0           ;GET CLEARING CONSTANT
0030  020B  A4 13              LDY PTR
0031  020D  D0 02              BNE DECR         ;IF PTR <> 0, DON'T DECREMENT HI BYTE.
0032  020F  C6 14              DEC PTR+1        ;DECREMENT HI BYTE OF POINTER.
0033  0211  C6 13       DECR   DEC PTR          ;DECREMENT LO BYTE.
0034  0213  81 13              STA (PTR,X)      ;CLEAR LOCATION.
0035  0215  A5 13              LDA PTR          ;CHECK IF POINTER = TABLE POINTER,
0036  0217  C5 10              CMP TABLE        ;IF UNEQUAL, CLEAR NEXT LOCATION.
0037  0219  D0 EE              BNE CLRLP
0038  021B  A5 14              LDA PTR+1
0039  021D  C5 11              CMP TABLE+1
0040  021F  D0 E8              BNE CLRLP
0041  0221  60                 RTS
0042  0222              ;
0043  0222              ;ROUTINE 'STORE': PLACES BUFFER CONTENTS IN
0044  0222              ;TABLE, USING 1ST 6 CHRS. OF BUFFER AS A
0045  0222              ;'KEY' TO DETERMINE HASHED ADDRESS IN
0046  0222              ;TABLE.
0047  0222              ;
0048  0222  A2 00       STORE  LDX #0           ;CLEAR X FOR INDEXED ADDRESSING.
0049  0224  20 90 02           JSR HASH         ;GET HASHED INDEX..
0050  0227  20 62 02     CMPR1  JSR LIMIT        ;MAKE SURE INDEX IS WITHIN BOUNDS.
0051  022A  A1 13              LDA (PTR,X)      ;CHECK DATA UNIT...
0052  022C  F0 05              BEQ EMPTY        ;JUMP IF EMPTY.
0053  022E  E6 12              INC INDX         ;TRY NEXT UNIT.
0054  0230  4C 27 02           JMP CMPR1        ;CHECK FOR NEXT UNIT INDEX VALID.
0055  0233  A0 07       EMPTY  LDY #7           ;LOOP BX TO LOAD DATA UNIT.
0056  0235  B9 16 00     FILL   LDA BUFFER,Y     ;GET CHR FROM BUFFER,
0057  0238  91 13              STA (PTR),Y      ;PLACE IT IN BUFFER.
0058  023A  88                 DEY
0059  023B  10 F8              BPL FILL         ;XFER NEXT CHR.
0060  023D  60                 RTS              ;ADDITION DONE.
0061  023E              ;
0062  023E              ;ROUTINE 'FIND' :
0063  023E              ;FINDS ENTRY WHOSE KEY IS IN BUFFER.
0064  023E              ;ENTRY, WHEN FOUND, IS COPIED INTO
0065  023E              ;BUFFER, ALONG WITH 2 BYTES OF DATA.
0066  023E              ;
0067  023E  A2 00       FIND   LDX #0           ;CLEAR X FOR INDIRECT ADDRESSING.
0068  0240  20 90 02           JSR HASH         ;GET HASH PRODUCT.
0069  0243  20 62 02     CMPR2  JSR LIMIT        ;MAKE SURE RESULT IS WITHIN LIMITS
```

Fig. 9-45 : Programme de hachage

```
0070  0246  A0 05          LDY #5          ;LOOP 6X TO COMPARE BUFFER TO DATA ITEM.
0071  0248  B1 13    CHKLP LDA (PTR),Y     ;GET CHR FROM TABLE.
0072  024A  D9 16 00       CMP BUFFER,Y    ;IS IT = BUFFER CHR?
0073  024D  D0 0E          BNE BAD         ;IF NOT, TRY NEXT DATA UNIT.
0074  024F  88             DEY
0075  0250  10 F6          BPL CHKLP       ;CHECK NEXT CHRS.
0076  0252  A0 07    MATCH LDY #7          ;LOOP 8X TO XFER CHRS TO BUFFER.
0077  0254  B1 13    XFER  LDA (PTR),Y     ;GET CHR. FROM TABLE.
0078  0256  99 16 00       STA BUFFER,Y    ;STORE IN BUFFER.
0079  0259  88             DEY
0080  025A  10 F8          BPL XFER        ;LOOP TO XFER CHRS.
0081  025C  60             RTS             ;DONE :DATA UNIT FOUND, IN BUFFER.
0082  025D  E6 12    BAD   INC INDX        ;NOT FOUND, TRY NEXT DATA UNIT.
0083  025F  4C 43 02       JMP CMPR2       ;VALIDATE NEW DATA UNIT INDEX.
0084  0262                ;
0085  0262                ;ROUTINE TO MAKE SURE DATA INDEX IS WITHIN
0086  0262                ;BOUNDS SET BY ENTNUM, THEN MULTIPLY INDEX
0087  0262                ;BY 8, AND ADD IT TO TABLE POINTER. THE
0088  0262                ;RESULT IS PLACED IN 'PTR' AS DATA UNIT ADDRESS.
0089  0262                ;
0090  0262  A5 12    LIMIT LDA INDX        ;GET INDEX.
0091  0264  C5 15    TEST  CMP ENTNUM      ;INDEX > NUMBER OF DATA ITEMS?
0092  0266  90 06          BCC OK`         ;JUMP IF NOT.
0093  0268  38             SEC             ;YES -
0094  0269  E5 15          SBC ENTNUM      ;SUBTRACT # OF ITEMS UNTIL
0095  026B  4C 64 02       JMP TEST        ;INDEX WITHIN BOUNDS.
0096  026E  85 13    OK    STA PTR         ;STORE GOOD INDEX IN POINTER.
0097  0270  85 12          STA INDX        ;SAVE UPDATED INDEX.
0098  0272  A9 00    SHADD LDA #0          ;CLEAR UPPER POINTER FOR SHIFT.
0099  0274  85 14          STA PTR+1
0100  0276  06 13          ASL PTR         ;SHIFT PTR 3X LEFT - MULTIPLY BY 8.
0101  0278  26 14          ROL PTR+1
0102  027A  06 13          ASL PTR
0103  027C  26 14          ROL PTR+1
0104  027E  06 13          ASL PTR
0105  0280  26 14          ROL PTR+1
0106  0282  18             CLC
0107  0283  A5 10          LDA TABLE       ;ADD POINTER AND TABLE START
0108  0285  65 13          ADC PTR         ;ADDRESS AND PLACE RESULT IN POINTER.
0109  0287  85 13          STA PTR
0110  0289  A5 11          LDA TABLE+1
0111  028B  65 14          ADC PTR+1
0112  028D  85 14          STA PTR+1
0113  028F  60             RTS
0114  0290                ;
0115  0290                ;ROUTINE TO GENERATE DATA UNIT INDEX IN TABLE
0116  0290                ;BY HASHING 'KEY', OR CHRS OF LABEL.
0117  0290                ;
0118  0290  A9 00    HASH  LDA #0          ;CLEAR LOCATION FOR INDEX.
0119  0292  18             CLC             ;PREPARE TO ADD.
0120  0293  A0 05          LDY #5          ;LOOP 6X FOR EXCLUSIVE ORS.
0121  0295  59 16 00 EXOR  EOR BUFFER,Y    ;EXCLUSIVE-OR ACCUM. WITH BUFFER CHR.
0122  0298  2A             ROL A           ;MULTIPLY ACCUM. BY 2.
0123  0299  88             DEY             ;COUNT DOWN CHRS.
0124  029A  10 F9          BPL EXOR        ;GET NEXT CHR.
0125  029C  85 12          STA INDX SAVE HASH PRODUCT AS INDEX.
0126  029E  60             RTS             ;DONE.
0127  029F                .END

ERRORS = 0000 <0000>

SYMBOL TABLE

SYMBOL   VALUE

BAD      025D   BUFFER   0016   CHKLP    0248   CLRLP    020F
CMPR1    0227   CMPR2    0243   DECR     0211   EMPTY    0233
ENTNUM   0015   EXOR     0295   FILL     0235   FIND     023E
HASH     0290   INDX     0012   INIT     0200   LIMIT    0262
MATCH    0252   OK       026E   PTR      0013   SHADD    0272
STORE    0222   TABLE    0010   TEST     0264   XFER     0254

END OF ASSEMBLY
```

Fig. 9-45 : Programme de hachage (suite)

TRI A BULLE

Le tri à bulle est une technique opérant la mise en ordre croissant ou décroissant des éléments d'une table. Il tire son nom du fait que l'élément le plus petit monte progressivement vers le haut de la table, comme une bulle de gaz. Chaque fois qu'il rencontre un élément plus lourd, il le saute. La Figure 9-47 en montre un exemple. La liste à trier comprend les nombres : 10, 5, 0, 2 et 100, et doit être mise en ordre croissant (0 en haut). L'algorithme est simple, et l'ordinogramme apparaît Figure 9-46.

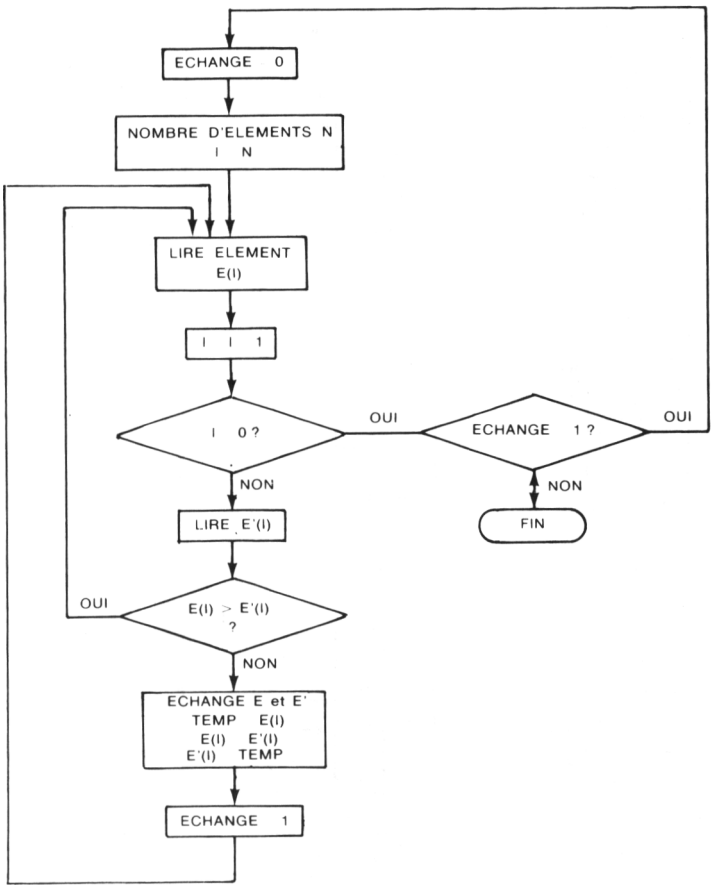


Fig. 9-46 : Tri à bulle

Les deux éléments du haut (ou du bas) sont comparés. Si le plus bas est inférieur (« plus léger »), ils sont échangés. Sinon, ils sont laissés tels quels. Pour des raisons pratiques, l'échange éventuel va être mémorisé, jusqu'à la prochaine paire d'éléments, etc... Jusqu'à ce que tous les éléments aient été comparés deux à deux.

Cette première passe est illustrée par les étapes 1, 2, 3, 4, 5 et 6 de la Figure 9-47 (de bas en haut ; on aurait pu, tout aussi bien, aller de haut en bas).

Si aucun échange n'a eu lieu, le tri est terminé. S'il y a eu au moins un échange, on recommence au moins une passe.

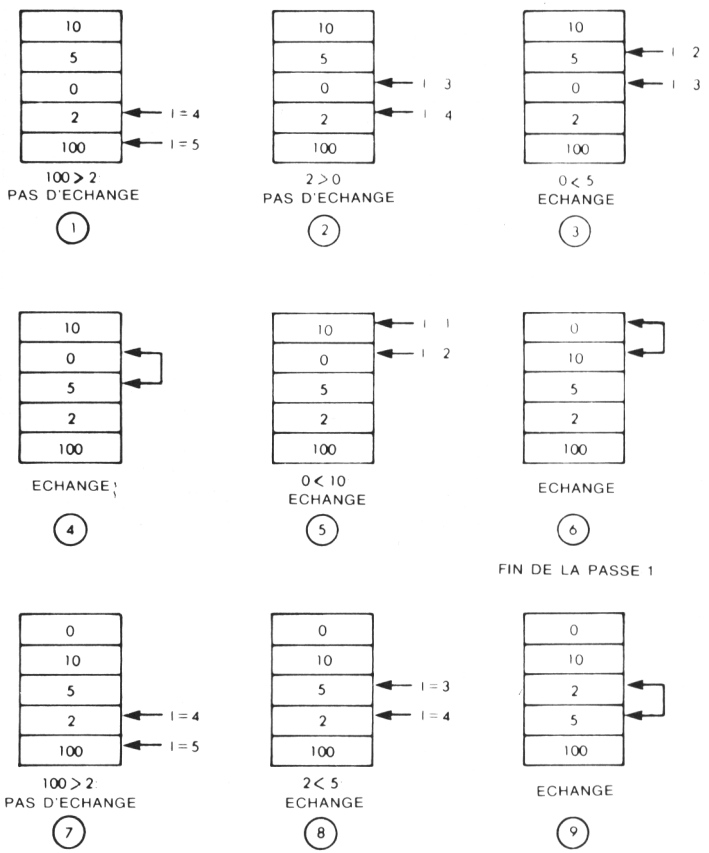
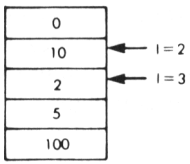
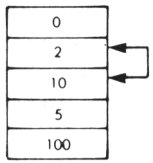


Fig. 9-47 : Exemple de tri à bulle



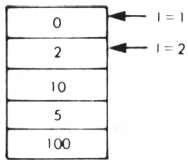
2<10:
ECHANGE

(10)



ECHANGE

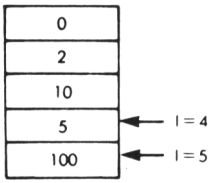
(11)



2>0
PAS D'ECHANGE

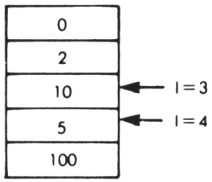
(12)

FIN DE LA PASSE 2



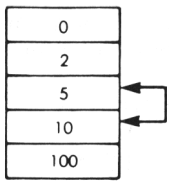
100>5:
PAS D'ECHANGE

(13)



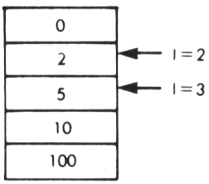
5<10:
ECHANGE

(14)



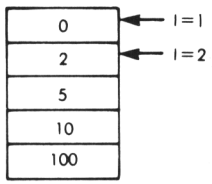
ECHANGE

(15)



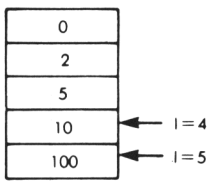
5>2:
PAS D'ECHANGE

(16)



2>0:
PAS D'ECHANGE

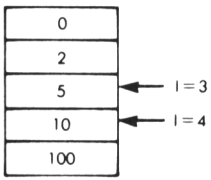
(17)



100>10:
PAS D'ECHANGE

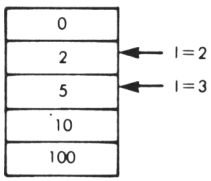
(18)

FIN DE LA PASSE 3



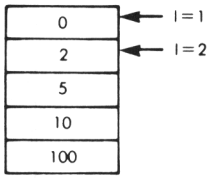
10>5:
PAS D'ECHANGE

(19)



5>2:
PAS D'ECHANGE

(20)



2>0:
PAS D'ECHANGE

(21)

FIN

D'après la Figure 9-47, nous voyons qu'au moins 4 passes sont nécessaires, dans cet exemple. Le processus décrit ci-dessus est simple, et largement utilisé. Une complication supplémentaire surgit à propos du mécanisme d'échange. En effet, pour échanger A et B, il est interdit d'écrire :

A = B
B = A

Cela nous ferait perdre la valeur précédente de A (essayez sur un exemple).

La solution correcte est d'utiliser une variable (ou une mémoire) temporaire, pour préserver la valeur de A :

TEMP = A
A = B
B = TEMP

Ça marche (là encore vérifiez-le sur un exemple). On appelle cette opération une permutation circulaire. C'est la manière dont tous les programmes réalisent l'échange. La technique est illustrée sur l'ordinogramme de la Figure 9-46.

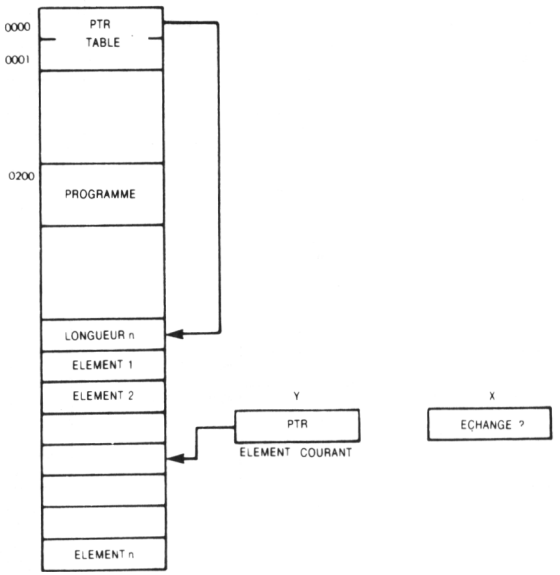


Fig. 9-48 : Tri à bulle : implantation mémoire

La carte d'implantation-mémoire correspondant au programme de tri à bulle apparaît Figure 9-48. Dans ce programme, chaque élément est un nombre positif de 8 bits. Le programme réside aux adresses 200 et suivantes. Le registre X mémorise le fait qu'un échange a eu, ou n'a pas eu, lieu, et le registre Y sert de pointeur courant dans la table. TAB est l'adresse de début de table. Le programme apparaît Figure 9-49. L'adressage indirect indexé, qui permet des accès efficaces, est utilisé partout.

Notez que la brièveté du programme est due à l'efficacité de l'adressage indirect du 6502.

```
SORT.....PAGE 0001

LINE # LOC     CODE       LINE

0002  0000              ;     BUBBLE SORT PROGRAM
0003  0000              ;
0004  0000                    * = $0
0005  0000              ;
0006  0000  00 06       TAB   .WORD $600
0007  0002              ;
0008  0002                    * = $200
0009  0200              ;
0010  0200  A2 00       SORT  LDX #0          ;SET  EXCHANGED  TO 0
0011  0202  A1 00             LDA (TAB,X)
0012  0204  A8                TAY             ;NUMBER OF ELEMENTS IS IN Y
0013  0205  B1 00       LOOP  LDA (TAB),Y     ;READ ELEMENT E(I)
0014  0207  88                DEY             ;DECREMENT NUMBER OF ELEMENTS TO READ.
0015  0208  F0 12             BEQ FINISH      ;END IF NO MORE ELEMENTS
0016  020A  D1 00             CMP (TAB),Y     ;COMPARE TO E(I)
0017  020C  B0 F7             BCS LOOP        ;GET NEXT ELEMENT IF E(I) E (I)
0018  020E  AA          EXCH  TAX             ;EXCHANGE ELEMENTS
0019  020F  B1 00             LDA (TAB),Y
0020  0211  C8                INY
0021  0212  91 00             STA (TAB),Y
0022  0214  8A                TXA
0023  0215  88                DEY
0024  0216  91 00             STA (TAB),Y
0025  0218  A2 01             LDX #1          ;SET  EXCHANGED  TO 1
0026  021A  D0 E9             BNE LOOP        ;GET NEXT ELEMENT
0027  021C  8A          FINISH TXA            ;SHIFT  EXCHANGED  TO A REG. FOR COMPARE...
0028  021D  D0 E1             BNE SORT        ;IF SOME EXCHANGES MADE, DO ANOTHER PASS.
0029  021F  60                RTS
0030  0220                    .END

ERRORS = 0000 <0000>



SYMBOL TABLE

SYMBOL   VALUE

EXCH     020E   FINISH   021C   LOOP     0205   SORT     0200
TAB      0000
END OF ASSEMBLY

<
<
```

Fig. 9-49 : Programme de tri à bulle

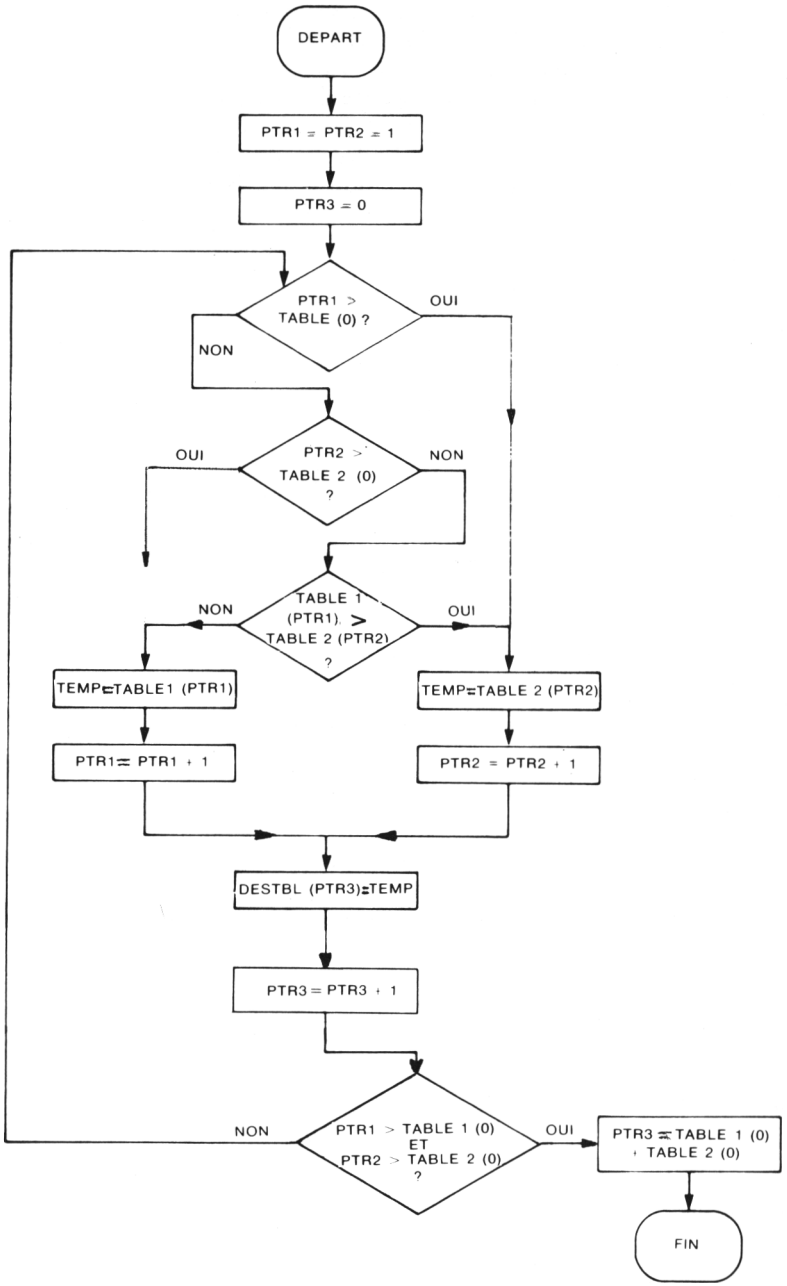


Fig. 9-50 : Ordinogramme de la fusion

ALGORITHME DE FUSION

Un autre problème usuel consiste à réunir deux ensembles de données en un seul. Nous supposerons ici que deux tables de données ont été préalablement classées, et qu'il s'agit de les réunir en une troisième. La longueur de chaque table de départ sera limitée à 256 octets (une page). Le premier élément de chaque table contient la longueur de la table.

L'algorithme de fusion apparaît Figure 9-50. L'organisation correspondante de la mémoire est donnée Figure 9-51, et le programme apparaît Figure 9-52. N'oubliez pas d'initialiser TABLE1, TABLE2 et DESTBL, avant d'utiliser le programme. L'algorithme lui-même est facile. Deux pointeurs courants, PTR1 et PTR2, pointent vers les tables de départ, et PTR3 vers la table d'arrivée.

Comparons les éléments courants de TABLE1 et TABLE2. La plus petite est copiée dans DESTBL, et le pointeur correspondant est incrémenté. Le processus est répété, et se terminera lorsque, à la fois, PTR1 et PTR2 auront atteint le bas de la table correspondante.

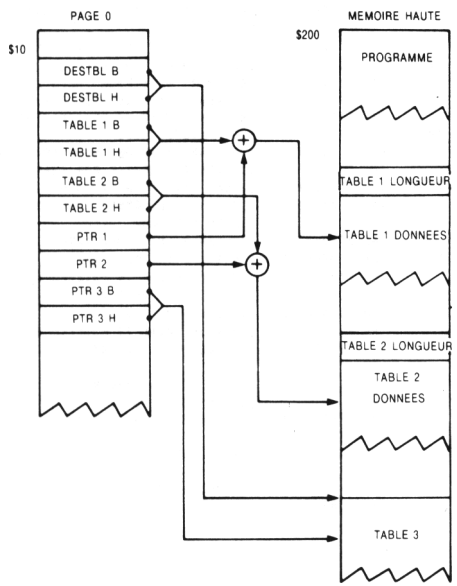


Fig. 9-51 : Implantation-mémoire pour la fusion

```
LINE # LOC     CODE       LINE

0002  0000              ;2-PAGE MERGE.
0003  0000              ;TAKES 2 DATA TABLES PREVIOUSLY SORTED,
0004  0000              ;AND MERGES THEM INTO A THIRD TABLE.
0005  0000              ;EACH SOURCE TABLE CAN BE UP TO ONE
0006  0000              ;PAGE (256 BYTES) IN LENGTH.
0007  0000              ;THE FIRST ELEMENT OF THE SOURCE
0008  0000              ;TABLES MUST CONTAIN THE TABLE LENGTH.
0009  0000              ;'PTR3' CONTAINS THE LENGTH OF THE
0010  0000              ;DESTINATION TABLE AT RETURN.
0011  0000              ;
0012  0000                    * = $10
0013  0010              DESTBL *=*+2          ;POINTER TO BEGINNING OF DESTINATION TABLE.
0014  0012              TABLE1 *=*+2          ;POINTER TO SOURCE TABLE 1.
0015  001A              TABLE2 *=*+2          ;POINTER TO SOURCE TABLE 2.
0016  0016              PTR1   *=*+1        ;TABLE 1 INDEX.
0017  0017              PTR2   *=*+1        ;TABLE 2 INDEX.
0018  0018              PTR3   *=*+2        ;DESTINATION TABLE INDEX.
0019  001A              ;
0020  001A                    * = $200
0021  0200              ;
0022  0200  A5 11              LDA DESTBL+1    ;PTR3 = TABLE3
0023  0202  85 19              STA PTR3+1
0024  0204  A5 10              LDA DESTBL
0025  0206  85 18              STA PTR3
0026  0208  A9 01              LDA #1          ;SET SOURCE TABLE POINTERS TO BEGINNING,
0027  020A  85 16              STA PTR1        ;SKIPPING TABLE LENGTHS.
0028  020C  85 17              STA PTR2
0029  020E  A2 00              LDX #0          ;CLEAR X FOR INDIRECT ADDRESSING.
0030  0210  A1 14        COMPR LDA (TABLE2,X)  ;IS TABLE 2 LENGTH <
0031  0212  C5 17              CMP PTR2        ;TABLE 2 POINTER?
0032  0214  90 19              BCC TKTB1       ;IF YES, GET BYTE FROM TABLE 1.
0033  0216  A1 12              LDA (TABLE1,X)  ;IS TABLE 1 LENGTH <
0034  0218  C5 16              CMP PTR1        ;TABLE 1 POINTER?
0035  021A  90 0A              BCC TKTB2       ;IF YES, GET BYTE FROM TABLE 2
0036  021C  A4 16              LDY PTR1        ;GET POINTER FOR TABLE 1.
0037  021E  B1 12              LDA (TABLE1),Y  ;USE IT TO FETCH BYTE.
0038  0220  A4 17              LDY PTR2        ;GET POINTER FOR TABLE 2,
0039  0222  B1 14              CMP (TABLE2),Y  ;USE IT TO FIND BYTE TO COMPARE
0040  0224                     ;TO TABLE 1 BYTE.
0041  0224  90 09              BCC TKTB1       ;IF TABLE 1 BYTE LESS, TAKE IT.
0042  0226  A4 17        TKTB2 LDY PTR2        ;GET POINTER FOR TABLE 2.
0043  0228  B1 14              LDA (TABLE2),Y  ;GET NEXT BYTE FROM TABLE 2.
0044  022A  E6 17              INC PTR2        ;INCREMENT POINTER FOR TABLE 2.
0045  022C  4C 35 02           JMP STORE       ;GO STORE BYTE IN DESTINATION TABLE.
0046  022F  A4 16        TKTB1 LDY PTR1        ;GET POINTER 1...
0047  0231  B1 12              LDA (TABLE1),Y  ;AND USE IT TO GET BYTE FROM TABLE.
0048  0233  E6 16              INC PTR1        ;INCREMENT POINTER FOR TABLE 1.
0049  0235  81 18        STORE STA (PTR3,X)    ;STORE BYTE AT NEXT LOCATION IN TABLE 3
0050  0237  E6 18              INC PTR3        ;INCREMENT LO ORDER TABLE 3 POINTER.
0051  0239  D0 02              BNE CC          ;IF NO OVERFLOW, SKIP
0052  023B  E6 19              INC PTR3+1      ;INCREMENT HI ORDER TABLE 3 POINTER.
0053  023D  A1 12        CC    LDA (TABLE1,X)  ;IS TABLE 1 LENGTH GREATER
0054  023F  C5 16              CMP PTR1        ;THAN OR EQUAL TO POINTER 1?
0055  0241  B0 CD              BCS COMPR       ;IF YES, GET NEXT BYTE.
0056  0243  A1 14              LDA (TABLE2,X)  ;IS TABLE 2 LENGTH GREATER
0057  0245  C5 17              CMP PTR2        ;THAN OR EQUAL TO POINTER 2?
0058  0247  B0 C7              BCS COMPR       ;IF YES, GET NEXT BYTE.
0059  0249  A9 00              LDA #0
0060  024B  85 19              STA PTR3+1      ;CLEAR PTR3 HI ORDER.
0061  024D  18                 CLC             ;MERGE DONE, NOW..
0062  024E  A1 12              LDA (TABLE1,X)  ;ADD TABLE 1 AND 2 LENGTHS.
0063  0250  61 14              ADC (TABLE2,X)
0064  0252  85 18              STA PTR3        ;STORE SUM IN TABLE 3 TEMPORARY POINTER.
0065  0254  90 04              BCC CCC         ;AND..
0066  0256  A9 01              LDA #1          ;OVERFLOW IN...
0067  0258  85 19              STA PTR3+1      ;HI BYTE.
0068  025A  60           CCC   RTS
0069  025B                    .END

ERRORS = 0000 <0000>
END OF ASSEMBLY
```

Fig. 9-52 : Programme de fusion

RECAPITULATION

Les concepts de base des structures de données usuelles ont maintenant été présentés, ainsi que des exemples de réalisation effective.

Grâce à ses modes d'adressage puissants, le 6502 se prête bien à la gestion de structures de données complexes. Son efficacité est démontrée par la concision des programmes que nous avons exposés.

En outre, nous avons présenté des techniques particulières de hachage, de tri et de fusion. Elles sont caractéristiques du type de techniques nécessaires à la résolution de problèmes complexes, mettant en jeu les véritables structures de données.

Le programmeur débutant pourra encore s'épargner les détails de la réalisation et de la gestion des structures de données.

Toutefois, une programmation efficace d'algorithmes non-triviaux, suppose une bonne compréhension des structures de données. Les exemples effectifs présentés dans ce chapitre, devraient permettre au lecteur d'acquérir cette compréhension, et de résoudre tous les problèmes habituels aux structures de données raisonnables.

10 CHAPITRE X

DEVELOPPEMENT DES PROGRAMMES

INTRODUCTION

Tous les programmes que nous avons étudiés et développés jusqu'ici l'ont été à la main, sans l'aide d'aucune ressource software ou hardware. La seule amélioration dont nous avons profité par rapport au codage purement binaire a été d'utiliser des symboles mnémoniques : ceux du langage assembleur. Pour un développement efficace de programmes, il est nécessaire de comprendre les diverses aides au développement hardware et software qui sont disponibles. Le but de ce chapitre est de les présenter et de les évaluer.

CHOIX FONDAMENTAUX DE LA PROGRAMMATION

Il y a trois possibilités fondamentales : écrire le programme en binaire ou hexadécimal, l'écrire en langage assembleur ou l'écrire en langage évolué. Passons-les en revue.

1. Codage hexadécimal

Un programme est habituellement rédigé en utilisant les mnémoniques du langage assembleur. Mais la plupart des systèmes bon marché en une seule carte ne possèdent pas d'assembleur (l'assembleur est le programme capable de traduire automatiquement les mnémoniques du programme dans les codes binaires voulus). Quand on ne dispose pas d'assembleur, cette traduction des mnémoniques en binaire doit être effectuée à la main. Le binaire est malcommode à utiliser et propice aux erreurs ; c'est pourquoi on utilise normalement l'hexadécimal dans un tel cas. Il a été vu au Chapitre 1

qu'un chiffre hexadécimal représente 4 bits en binaire. Deux chiffres hexadécimaux seront donc utilisés pour représenter le contenu de chaque octet. A titre d'exemple, une table donnant l'équivalent hexadécimal des instructions du 6502 figure en appendice.

Quand les ressources financières ou matérielles de l'utilisateur sont limitées et qu'il ne dispose pas d'un assembleur, il doit donc traduire le programme en hexadécimal à la main. Ceci est raisonnable pour un petit nombre d'instructions, tel que peut-être 10 à 100. Pour des programmes plus longs, ce processus est fastidieux et il favorise les erreurs, de sorte qu'on tend à l'éviter. Presque tous les microordinateurs en une seule carte nécessitent l'entrée des programmes en hexadécimal. Ils ne possèdent pas d'assembleur et ne sont pas équipés d'un clavier alphanumérique complet afin de limiter leur coût.

En résumé, le codage hexadécimal n'est pas une manière idéale d'introduire un programme dans un ordinateur. C'est simplement un moyen économique. Un compromis coût/efficacité est établi entre le prix de la configuration nécessaire pour l'utilisation d'un assembleur et celui du clavier alphanumérique nécessaire, et la tâche fastidieuse d'entrer le programme en mémoire en hexadécimal. Néanmoins cela ne change en aucun cas la manière d'écrire le programme proprement dit. *Le programme est quand même écrit en assembleur* afin de pouvoir être examiné facilement par le programmeur et d'avoir un sens pour le lecteur humain.

2. Programmation en langage assembleur

La programmation au niveau assembleur couvre à la fois les programmes qui peuvent être entrés en hexadécimal et ceux qui peuvent être introduits dans le système sous forme assembleur symbolique. Examinons maintenant l'entrée d'un programme directement dans sa représentation en langage assembleur. Nous supposerons disposer d'un assembleur. L'assembleur va lire chacune des instructions du programme sous forme mnémonique et la traduire dans le motif binaire voulu, utilisant 1, 2 ou 3 octets comme le spécifie le codage des instructions. En outre, un bon assembleur offre un certain nombre de facilités supplémentaires pour écrire le programme. Elles seront passées en revue ci-dessous dans la section consacrée à l'assembleur. En particulier, des *directives* sont disponibles pour modifier la valeur des symboles. Des *adresses symboliques* peuvent être utilisées, et on peut effectuer des branchements à des étiquettes symboliques. Au cours de la phase de mise au point, où l'utilisateur enlève, on ajoute, des instructions, il ne sera pas nécessaire de réécrire tout le programme si on insère une instruction entre un branchement et sa cible aussi longtemps que l'on utilise des adresses symboliques.

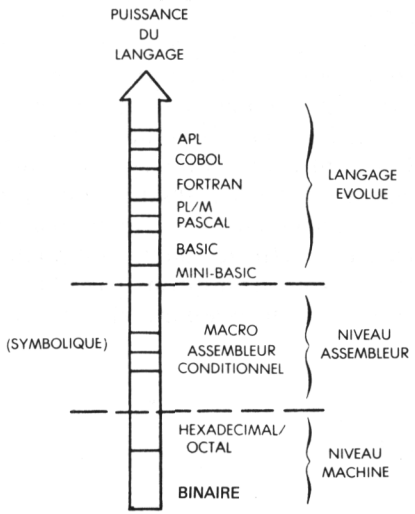


Fig. 10-1 : Les choix de programmation

L'assembleur se chargera d'ajuster automatiquement toutes les étiquettes lors du processus de traduction. De plus, l'assembleur permet à l'utilisateur de corriger son programme sous forme symbolique. On peut utiliser un *désassembleur* pour examiner le contenu de la mémoire, et reconstituer les instructions du niveau assembleur qu'elle contient.

Les différentes autres ressources software dont on dispose normalement dans un système seront passées en revue ci-dessous. Examinons maintenant la troisième possibilité de programmation (voir Figure 10-1).

3. Langage évolué (ou de haut niveau)

Le programme peut également être rédigé dans un langage évolué comme BASIC, APL, PASCAL ou autre.

Les techniques de programmation dans ces divers langages sont exposées dans des livres spécifiques (voir par exemple nos livres sur BASIC) et elles ne seront pas rappelées ici. Nous ne discuterons donc que brièvement de ce mode de programmation. Les langages évolués offrent des instructions puissantes qui rendent la programmation plus facile et plus rapide. Ces instructions doivent ensuite être traduites par un programme complexe dans la forme binaire définitive que, seule, un ordinateur peut exécuter. En général, chaque

instruction de haut niveau sera traduite en un grand nombre d'instructions binaires élémentaires. Le programme qui effectue cette traduction automatique s'appelle un *compilateur* ou un *interpréteur*. Un compilateur traduit en code objet toutes les instructions d'un programme à la suite. Le code résultant est alors exécuté dans une phase distincte. Par contraste, un interpréteur interprète une seule instruction à la fois et il l'exécute aussitôt, puis il « traduit » la suivante et l'exécute. L'interpréteur offre l'avantage d'une réponse dite interactive mais est moins efficace qu'un compilateur sur le plan de la vitesse d'exécution. Nous n'en dirons pas plus ici. Revenons à la programmation d'un microprocesseur réel au niveau langage assembleur.

AIDES LOGICIEL

Nous allons passer en revue ici les principales facilités software qui sont (ou devraient être) offertes par le système pour permettre le développement commode de logiciel. Certaines définitions ont déjà été présentées. Elles seront rappelées ici et les autres programmes fondamentaux vont être présentés avant de passer à la suite.

L'assembleur est le programme qui traduit les instructions sous forme mnémonique en leur équivalent binaire. Il traduit normalement une instruction symbolique en une instruction binaire (qui peut occuper 1, 2 ou 3 octets). Le code binaire résultant s'appelle *code objet*. Il est directement exécutable par le microordinateur. Comme sous-produit, l'assembleur fournit aussi un listing symbolique complet du programme ainsi que les tables d'équivalence à utiliser par le programmeur et la liste des occurrences de symboles dans le programme. Des exemples seront présentés plus loin dans ce chapitre.

Un compilateur est un programme qui traduit en binaire des instructions d'un langage évolué.

Un interpréteur est un programme analogue à un compilateur. Lui aussi traduit en binaire des instructions en langage évolué, mais il ne conserve pas la représentation intermédiaire et il exécute les instructions immédiatement. En fait, souvent il ne génère même pas de code intermédiaire mais exécute directement les instructions de haut niveau.

Le moniteur est le programme de base qui est nécessaire pour utiliser les ressources hardware du système. Il surveille continuellement les organes d'entrée et gère les autres périphériques éventuels. Par exemple un moniteur minimal pour microordinateur sur une carte,

muni d'un clavier et de LED, doit parcourir le clavier en permanence pour déceler une entrée de l'utilisateur, et aussi afficher les données voulues sur les diodes électroluminescentes. En outre, il doit être capable de comprendre un nombre limité de commandes issues du clavier, comme DEPART, ARRET, CONTINUER, CHARGER MEMOIRE, EXAMINER MEMOIRE. Sur un système plus important le moniteur est souvent appelé programme *exécutif,* lorsqu'une gestion complexe des fichiers et une synchronisation des tâches élaborées sont aussi présentes. L'ensemble de toutes les facilités est appelée *système d'exploitation.* Dans le cas où les fichiers résident sur disque, le système d'exploitation s'appelle *système d'exploitation disque* (DOS = disk operating system).

Un *éditeur* est un programme conçu pour faciliter l'entrée et la modification de textes ou de programmes. Il permet à l'utilisateur d'entrer des caractères de façon commode, d'en ajouter, de les insérer, d'ajouter des lignes, d'en supprimer, de rechercher des caractères ou des groupes. C'est une ressource importante pour entrer des textes de façon efficace et commode.

Un « dévermineur » (debugger) est une ressource nécessaire pour dépanner les programmes. Quand un programme ne marche pas correctement, il n'y a généralement aucune indication sur la cause, quelle qu'elle soit. Le programmeur veut par conséquent insérer des *points d'arrêt* (break-points) dans son programme, pour en suspendre l'exécution à des adresses spécifiques, afin de pouvoir examiner le contenu des registres ou de la mémoire en ces points. C'est la fonction principale d'un dévermineur. Le dévermineur permet ainsi de suspendre un programme, reprendre l'exécution, examiner, afficher et modifier le contenu des registres et de la mémoire. Un bon dévermineur est muni de facilités supplémentaires telles que la possibilité d'examiner les données sous forme symbolique, hexadécimale, binaire, ou toute autre représentation usuelle, ainsi que la possibilité d'entrer des données sous cette forme.

Le chargeur, ou *chargeur-éditeur de liens* place les différents blocs de code objet dans les positions-mémoire spécifiées et il ajuste leurs pointeurs symboliques respectifs de façon qu'ils puissent se référencer les uns les autres. Il sert à reloger les programmes ou les blocs dans des zones-mémoire différentes.

Un *simulateur* ou un *émulateur* sert à simuler le fonctionnement d'un organe, généralement le microprocesseur, en son absence, lorsqu'on développe un programme sur un processeur simulé avant de le placer sur la carte définitive.

Avec cette approche, il devient possible de suspendre le programme, de le modifier et de le garder en mémoire RAM, de

manière à pouvoir le modifier commodément. Les inconvénients du simulateur sont que :

1) Généralement il ne simule que le processeur proprement dit, pas les organes d'entrée-sortie.

2) La vitesse d'exécution est lente et on opère en temps simulé. Il n'est donc pas possible de simuler les organes en temps réel et il peut encore se révéler des problèmes de synchronisation sur le système définitif même si la logique du programme a été trouvée correcte.

Un *émulateur* est essentiellement un simulateur en temps réel. Il utilise un processeur pour en simuler un autre en détail complet.

Les programmes utilitaires sont essentiellement les routines qui sont nécessaires dans la plupart des applications (et, généralement, que l'utilisateur souhaiterait que le constructeur ait fournies !).

On peut y inclure la multiplication, la division, et les autres opérations arithmétiques, les routines de transfert de blocs, les tests de caractères, les routines de gestion des périphériques (« handlers » ou « drivers ») etc.

Le processus de développement d'un programme

Nous allons maintenant examiner une séquence typique de développement d'un programme au niveau assembleur. Nous supposerons, afin d'en démontrer l'intérêt, que toutes les ressources software usuelles sont disponibles. Si elles manquent dans un système donné, il reste possible de développer des programmes, mais avec moins de commodité, et par suite, le temps nécessaire pour mettre au point le programme en sera probablement accru.

L'approche habituelle est de concevoir d'abord un algorithme, et définir les structures de données convenant au problème à résoudre. Ensuite, il convient de développer un ensemble complet d'ordinogrammes qui représentent le déroulement du programme. Enfin les ordinogrammes sont traduits en langage assembleur pour le microprocesseur : c'est la phase de codage.

Le programme doit alors être entré dans l'ordinateur. Nous examinerons dans la prochaine section les options hardware à utiliser lors de cette phase. Le programme est mis en RAM du système sous contrôle de l'éditeur.

Lorsqu'une section du programme, formée d'un ou plusieurs sous-programmes, a été entrée, elle va être testée.

On va d'abord utiliser l'assembleur. Si l'assembleur ne réside pas déjà dans le système, on le charge à partir d'une mémoire externe telle qu'un disque. Ensuite le programme est assemblé, c'est-à-dire

traduit en binaire. Il en résulte un programme objet prêt à être exécuté.

On ne doit pas normalement espérer qu'un programme marche correctement la première fois. Pour vérifier son fonctionnement correct, on installe généralement un certain nombre de points d'arrêt aux points cruciaux, où il est facile de tester si les résultats intermédiaires sont corrects. On utilise pour cela le dévermineur. On spécifie les endroits choisis (les adresses) pour les points d'arrêt. On envoie alors une commande « GO », qui fait démarrer l'exécution.

Le programme s'arrête automatiquement à chacun des points d'arrêt spécifiés. Le programmeur peut alors vérifier en examinant le contenu des registres ou de la mémoire que les données sont correctes jusque-là. Si elles sont correctes, on continue jusqu'au prochain point d'arrêt. Chaque fois qu'on trouve une donnée fausse, on a identifié une erreur dans le programme. A ce moment le programmeur se reporte au listing de son programme et il vérifie si son codage a été correct. Si aucune erreur ne peut être trouvée dans la programmation, il peut s'agir d'une erreur logique et on se reporte à l'ordinogramme. Nous supposerons ici que les ordinogrammes ont été vérifiés à la main et qu'on peut raisonnablement supposer qu'ils sont corrects ; l'erreur provient vraisemblablement du codage. Il sera donc nécessaire de modifier une portion du programme.

Si la représentation symbolique du programme est toujours en mémoire, on va simplement « rappeler » l'éditeur et modifier les lignes voulues, puis revenir à la séquence précédente. Dans certains systèmes, la mémoire disponible peut ne pas être suffisante, de sorte qu'il est nécessaire de vider la représentation symbolique du programme sur un disque ou une cassette avant d'exécuter le code objet. Naturellement, dans ce cas, on aurait à recharger la représentation symbolique du programme à partir de son support avant de rappeler l'éditeur. La procédure précédente sera répétée autant de fois que nécessaire jusqu'à ce que les résultats du programme soient corrects.

Insistons sur le fait que mieux vaut prévenir que guérir. Une conception correcte conduira généralement à un programme qui fonctionnera correctement très vite après élimination des fautes de frappe habituelles ou des erreurs de codage évidentes. Au contraire, une conception peu nette peut entraîner des programmes dont la mise au point sera extrêmement longue. Le temps de dépannage est généralement considéré comme beaucoup plus long que le temps de conception proprement dite (souvent dix fois plus). En bref, il est toujours rentable d'investir plus de temps dans la conception pour raccourcir la phase du dépannage.

Néanmoins, cette approche permet bien de tester l'organisation globale du programme mais pas son fonctionnement en temps réel avec les périphériques. S'il faut tester les périphériques, la solution directe consiste à transférer le programme dans des EPROM (en général), et à l'installer sur la carte, puis observer s'il fonctionne. Il y a une meilleure solution. C'est l'utilisation d'un «*émulateur en circuit*» (« ICE »). Un tel dispositif fait appel à un 6502 (ou à tout autre microprocesseur) pour émuler le 6502 en temps (presque) réel. Il émule le 6502 physiquement. L'émulateur possède un câble terminé par un connecteur à 40 broches, exactement identique aux broches du 6502. Ce connecteur peut alors être inséré sur la carte d'application réelle qu'on est en train de développer. Les signaux générés par l'émulateur seront exactement identiques à ceux du 6502 excepté qu'ils seront un peu plus lents. L'avantage essentiel est que le pro-

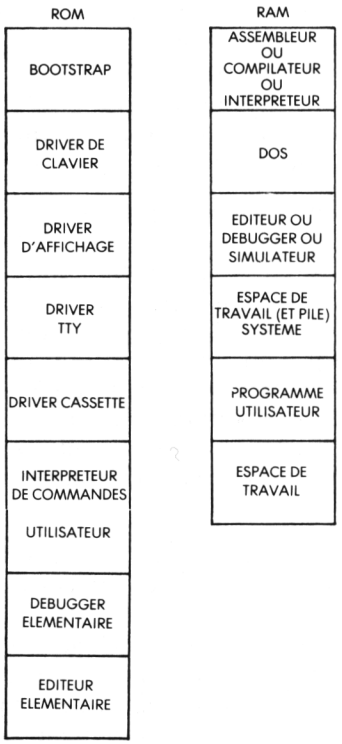


Fig. 10-2 : Carte d'implantation-mémoire type

gramme en cours de test résidera encore dans la RAM du système de développement. Il génèrera les signaux réels qui communiqueront avec les véritables périphériques qu'on souhaite utiliser. Il en résulte qu'il devient possible de continuer à développer le programme en utilisant toutes les ressources du système de développement (éditeur, dévermineur, programmes symboliques, système de fichiers) tout en testant les entrées-sorties en temps réel.

En outre, un bon émulateur offre des facilités particulières comme la possibilité de *trace*. La trace est un enregistrement des dernières instructions ou de l'état des différents bus du système avant un point d'arrêt.

En bref, la trace fournit le film des événements qui se sont produits avant le point d'arrêt ou la panne. On peut même déclencher un oscilloscope à une adresse déterminée ou en fonction de bits spécifiques. Une telle facilité est d'un grand intérêt car, quand on rencontre une erreur, il est généralement trop tard. L'instruction ou la donnée qui a causé l'erreur a été présente *avant* la détection. La possibilité d'une trace permet à l'utilisateur de trouver quelle section du programme a causé l'erreur. Si la trace n'est pas assez longue, on établit simplement un point d'arrêt en amont.

Ceci complète notre description de la séquence d'événements habituels mis en jeu dans le développement d'un programme. Passons maintenant en revue les possibilités hardware disponibles pour développer des programmes.

LES CHOIX HARDWARE

1. Microordinateur sur une carte

Le microordinateur sur une carte permet l'approche la moins coûteuse au développement des programmes. Il est généralement muni d'un clavier hexadécimal plus quelques touches de fonction et de six afficheurs à LED qui permettent de visualiser une adresse et la donnée correspondante. Comme il ne possède que peu de mémoire, il ne dispose généralement pas d'un assembleur. Au mieux, il a un moniteur simple et pratiquement pas de facilités d'édition ou de déverminage si ce n'est quelques commandes peu nombreuses. Tous les programmes doivent, par suite, être entrés sous forme hexadécimale. Ils seront aussi visualisés en hexadécimal sur les LED. Un microordinateur sur une carte a, en théorie, la même puissance hardware que n'importe quel ordinateur. Simplement, comme sa taille-mémoire et son clavier sont restreints, il n'a pas toutes les facilités habituelles des systèmes plus grands et il en résulte un temps de développement des programmes plus long. Comme il est fastidieux de développer

des programmes en hexadécimal, le microordinateur sur une carte est surtout adapté à la formation et l'entraînement, où on n'a à développer que des programmes de longueur limitée et où cette longueur réduite n'est pas un obstacle. Les cartes autonomes constituent probablement la méthode la moins coûteuse d'apprentissage actif de la programmation. Néanmoins, ils ne peuvent servir au développement de programmes complexes, à moins qu'on ne leur connecte des cartes-mémoire supplémentaires et qu'on ne rende disponibles les aides software habituelles.

2. Le système de développement

Un système de développement est un microordinateur muni d'une RAM de taille suffisante (32K ou 48K) ainsi que des périphériques nécessaires, tels qu'un écran, une imprimante, deux disques et généralement un programmateur de PROM, plus peut-être, un émulateur-en-circuit. Le système de développement est spécialement prévu pour faciliter le développement de programmes en milieu industriel.

Il offre généralement toutes les facilités software mentionnées dans la section précédente, ou la plupart d'entre elles. En principe, c'est l'outil idéal pour le développement de software.

La limitation du système de développement basé sur un microordinateur est qu'il n'est peut être pas capable d'avoir un compilateur ou un interpréteur. C'est parce qu'un compilateur exige généralement une très grande quantité de mémoire, souvent plus qu'il n'y en a dans tout le système. Toutefois, pour développer des programmes au niveau assembleur, il offre toutes les facilités nécessaires. Néanmoins, comme les systèmes de développement se vendent en nombre relativement faible si on compare avec les ordinateurs individuels, ils coûtent considérablement plus cher.

3. Microordinateurs individuels

Le hardware d'un microordinateur individuel est, naturellement, exactement le même que celui d'un système de développement. La principale différence réside dans le fait qu'il n'est normalement pas muni des aides au développement software perfectionnées qu'on trouve sur un système de développement industriel. Par exemple, de nombreux microordinateurs individuels offrent seulement un assembleur élémentaire, un éditeur minimal, un système de fichiers minimal ; ils n'ont pas de dispositif pour connecter un programmateur de PROM, ni d'émulateur-en-circuit, ni de dévermineur puissant. Ils représentent donc un échelon intermédiaire entre le microordinateur sur une seule carte et le système de développement complet. Pour un utilisateur qui souhaite développer des pro-

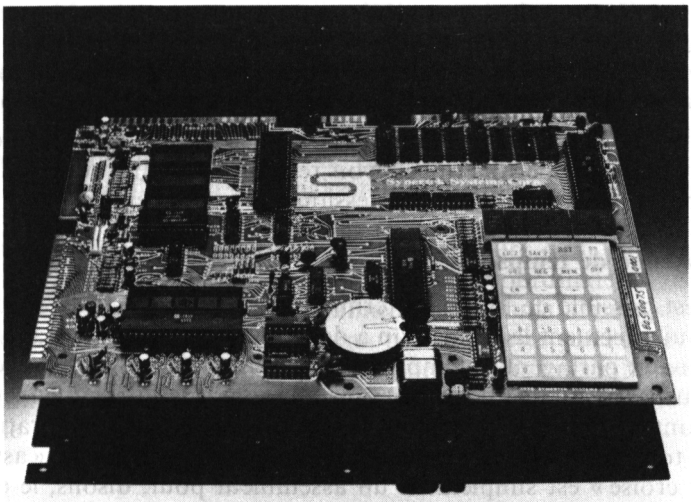


Fig. 10-3 : Le SYM.1 est un microordinateur sur une carte typique

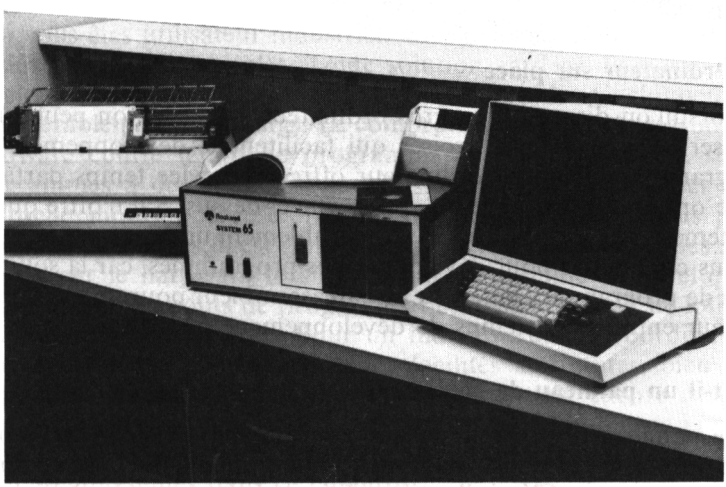


Fig. 10-4 : Le System 65 de Rockwell/Synertek est un système de développement

grammes de complexité modérée, ils constituent probablement le meilleur compromis puisqu'ils offrent l'avantage d'un prix bas tout en ayant un ensemble raisonnable d'outils de développement même s'ils sont assez limités du point de vue de la commodité d'emploi.

Pour une description complète et détaillée des ordinateurs individuels et de leurs périphériques, voir réf. C1, « Une Introduction aux Ordinateurs Individuels et Professionnels », du même auteur.

4. Système de temps partagé

Il est possible de louer des terminaux auprès de sociétés spécialisés, qui effectuent la connexion à des réseaux de temps partagé. Ces terminaux partagent le temps de gros ordinateurs et ils bénéficient de tous les avantages de grandes installations. Des *assembleurs croisés* sont disponibles pour tous les microprocesseurs sur pratiquement tous les systèmes commerciaux de temps partagé. Un « assembleur croisé » est simplement un assembleur pour, disons, le 6502, mais qui tourne, par exemple, sur IBM 370. Formellement, un assembleur croisé est un assembleur pour le microprocesseur X qui tourne sur le processeur Y. L'ordinateur utilisé est sans importance. L'utilisateur écrit toujours son programme en langage assembleur de 6502 et l'assembleur croisé le transforme en motif binaire approprié. La différence, est toutefois, que le programme ne peut pas être exécuté à ce niveau. Il peut être exécuté par un simulateur, si on en dispose, à condition qu'il n'utilise aucune ressource d'entrées-sorties. Cette solution n'est donc utilisée qu'en environnement industriel.

5. Ordinateur sur place

Lorsqu'on dispose d'un gros ordinateur sur place, on peut aussi utiliser les assembleurs croisés qui facilitent le développement de programmes. Si un tel ordinateur offre un service temps partagé, cette option est analogue à l'option vue ci-dessus. S'il n'offre que le traitement par lots (batch), c'est probablement une des méthodes les moins commodes pour développer des programmes, car la soumission de programmes batch au niveau assembleur pour un microprocesseur entraîne un temps de développement très long.

Faut-il un panneau de commande ?

Le panneau de commande (ou panneau avant) est un accessoire hardware souvent utilisé dans le passé pour faciliter la mise au point des programmes. C'était l'outil traditionnel et commode d'affichage du contenu binaire d'un registre ou d'une mémoire. Néanmoins, toutes les fonctions du panneau de commande peuvent être accom-

plies à partir d'un terminal et la prédominance actuelle des écrans permet d'obtenir un service presque équivalent au panneau de commande : il suffit d'afficher la valeur binaire des bits. L'avantage supplémentaire de l'utilisation de l'écran est qu'on peut à volonté passer de la représentation binaire à l'hexadécimal, au symbolique ou au décimal, à condition bien sûr, de disposer des routines de conversion. L'inconvénient du terminal est que l'on doit taper sur plusieurs touches avant d'obtenir l'affichage voulu, au lieu de tourner un bouton. Cependant, comme le coût de la fourniture d'un panneau de commande est substantiel, la plupart des microordinateurs récents l'ont abandonné. L'intérêt du panneau de commande est souvent évalué en fonction d'arguments émotionnels basés sur l'expérience passée de l'individu plutôt que en fonction d'arguments rationnels ce qui donne lieu à des controverses passionnées. Le panneau de commande n'est pas indispensable.

RECAPITULATION DES RESSOURCES HARDWARE

Trois grandes catégories doivent être distinguées : si vous n'avez qu'un budget limité et souhaitez apprendre à programmer, achetez un microordinateur sur une carte. Son utilisation vous permettra de développer tous les programmes simples de ce livre et beaucoup d'autres. Néanmoins par la suite, quand vous voudrez développer des programmes de plus de quelques centaines d'instructions, vous ressentirez les limitations de cette approche.

Si vous êtes utilisateur industriel, vous avez besoin d'un système de développement complet. Toute solution en deçà du système de développement complet mènera à des temps de développement considérablement plus longs. Le compromis est évident : ressources hardware contre temps de programmation. Naturellement, si les programmes à développer sont très simples, on peut utiliser une approche moins onéreuse. Mais, s'il y a à développer des programmes complexes, il est généralement difficile de justifier des économies sur le hardware lors de l'achat du système de développement, vu que les coûts de programmation constitueront, de loin, le coût dominant du projet. Pour un individu, un microordinateur individuel offrira généralement des facilités suffisantes, bien que minimales. Un bon logiciel de développement est encore à venir pour la plupart des ordinateurs individuels, mais commence à apparaître. L'utilisateur devra juger son système en fonction des commentaires présentés dans ce chapitre.

Analysons maintenant plus en détail la ressource la plus indispensable : l'assembleur.

L'ASSEMBLEUR

Nous avons utilisé le langage assembleur tout le long de ce livre, sans en présenter la définition et la syntaxe formelles. Il est temps de le faire. Le langage assembleur est conçu pour permettre une représentation symbolique commode du programme de l'utilisateur tout en facilitant au programme assembleur la tâche de convertir ces mnémoniques en binaire.

Champs de l'assembleur

Nous avons vu que lorsqu'on tape un programme pour l'assembleur, on utilise différents champs qui sont :

Le champ étiquette, facultatif, qui peut contenir une adresse symbolique attribuée à l'instruction qui suit.

Le champ instruction qui comprend le code opération et les opérandes (on peut séparer le champ opérande).

Le champ commentaire, le plus à droite, qui est facultatif et permet d'expliquer le programme.

Une fois que le programme a été fourni à l'assembleur, celui-ci en produit un *listing.* Lors de la génération du listing, l'assembleur ajoute trois champs, généralement dans la partie gauche de la page. Un exemple apparaît ci-dessous : tout à fait à gauche, figure le numéro de ligne. Chaque ligne tapée par le programmeur reçoit un numéro de ligne symbolique. Le champ qui suit à droite est le champ adresse réelle qui montre la valeur hexadécimale du compteur ordinal lorsqu'il pointera sur l'instruction considérée.

Le champ suivant contient la représentation hexadécimale de l'instruction. Ceci montre l'une des utilisations possibles de l'assembleur : même si on prépare des programmes pour un microordinateur en une seule carte qui n'accepte que l'hexadécimal, on doit toujours écrire le programme au niveau assembleur, en s'efforçant d'avoir accès à un système muni d'un assembleur. A ce moment, on fait tourner les programmes sur ce système, en faisant appel à l'assembleur. L'assembleur générera automatiquement le code hexadécimal convenable, qu'on n'a plus qu'à taper sur son propre système. Ceci montre, sur un exemple simple, l'intérêt des ressources software supplémentaires.

Tables

Quand l'assembleur traduit le programme symbolique en binaire il accomplit deux tâches principales :

1) Il traduit les codes opération mnémoniques en leur équivalent binaire.

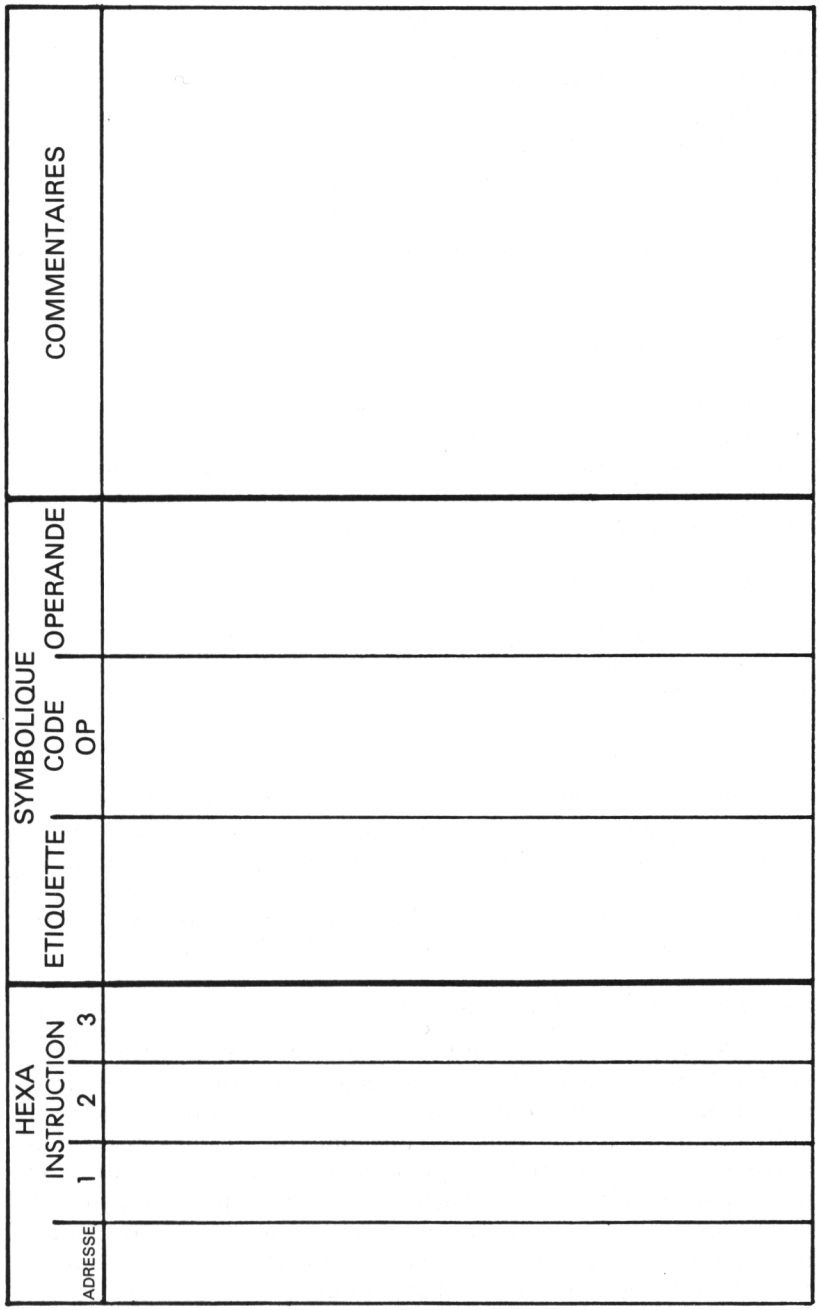


Fig. 10-5 : Formulaire de programmation pour microprocesseurs

2) Il traduit les symboles utilisés pour les constantes et les adresses en leur équivalent binaire.

Pour faciliter le dépannage du programme, l'assembleur imprime à la fin du listing la correspondance entre les symboles utilisés et leur valeur binaire. Cela s'appelle la table des symboles.

Certaines tables des symboles ne listent pas seulement le symbole et sa valeur mais aussi les numéros de ligne où le symbole apparaît : c'est une commodité supplémentaire.

Messages d'erreur

Lors du processus de l'assemblage, l'assembleur détecte des erreurs de syntaxe et il les liste dans le listing final. Les diagnostics typiques comprennent les symboles non définis, les étiquettes doublement définies, les codes opération illégaux, les adresses illégales, les modes d'adressage interdits. Beaucoup d'autres diagnostics de détail sont naturellement désirables, et ils sont généralement fournis. Ils varient avec chaque assembleur.

Le langage assembleur

Les codes opération ont déjà été définis. Nous allons ici définir les symboles, constantes et opérateurs qu'on peut utiliser dans la syntaxe de l'assembleur.

Symboles

On utilise des symboles pour représenter des valeurs numériques, données ou adresses. Traditionnellement, un symbole peut avoir 6 caractères dont le premier doit être alphabétique. Il y a une restriction supplémentaire : les 56 codes opération du 6502 et les noms des registres, c'est-à-dire A, X, Y, S, P. PC n'ont pas le droit d'être utilisés comme symboles utilisateur, car ils sont utilisés par l'assembleur.

Affectation d'une valeur à un symbole

Les étiquettes sont des symboles spéciaux dont la valeur n'a pas besoin d'être définie par le programmeur. Elles correspondent automatiquement à l'adresse de la ligne où elles apparaissent. Mais les autres symboles utilisés pour les constantes ou les adresses-mémoire doivent être définis par le programmeur avant leur utilisation. On utilise pour cela un signe « égale » ou bien une « directive » spéciale. C'est une instruction pour l'assembleur, qui ne sera pas traduite en instruction exécutable : on appelle cela une *directive* d'assemblage. Par exemple, la constante ALPHA pourrait être définie ainsi :

```
ALPHA = $A000
```

```
LINE # LOC     CODE       LINE

0057  0342  A9 00          LDA #$00
0058  0344  8D 0B A0       STA ACR1        ;TURN BOTH TIMERS OFF
0059  0347  8D 0B AC       STA ACR2
0060  034A  A2 20          LDX #OFFDEL     ;GET TONES-OFF DELAY CONSTANT
0061  034C  20 55 03  OFF   JSR DELAY      ;DELAY WHILE TONE IS OFF
0062  034F  CA             DEX
0063  0350  D0 FA          BNE OFF
0064  0352  4C 02 03       JMP DIGIT       ;GO BACK FOR NEXT DIGIT OF PHONE NUMBER
0065  0355                 ;
0066  0355                 ;THIS IS A SIMPLE DELAY ROUTINE FOR THE TONE ON AND OFF PERI
0067  0355                 ;
0068  0355  A9 FF          DELAY  LDA #DELCON     ;GET DELAY CONSTANT
0069  0357  38             WAIT   SEC             ;DELAY FOR THAT LONG
0070  0358  E9 01                  SBC #$01
0071  035A  D0 FB                  BNE WAIT
0072  035C  60                     RTS
0073  035D                 ;
0074  035D                 ;THIS IS A TABLE OF THE CONSTANTS FOR THE TONE FREQUENCIES
0075  035D                 ;FOR EACH TELEPHONE DIGIT.  THE CONSTANTS ARE TWO BYTES
0076  035D                 ;LONG, LOW BYTE FIRST.
0077  035D                 ;
0078  035D  13             TABLE  .BYTE $13,$02,$76,$01 ;TWO TONES FOR '0'
0078  035E  02
0078  035F  76
0078  0360  01
0079  0361  CD                    .BYTE $CD,$02,$9E,$01 ;TWO TONES FOR '1'
0079  0362  02
0079  0363  9E
0079  0364  01
0080  0365  CD                    .BYTE $CD,$02,$76,$01 ;      '2'
0080  0366  02
0080  0367  76
0080  0368  01
0081  0369  CD                    .BYTE $CD,$02,$53,$01 ;           '3'
0081  036A  02
0081  036B  53
0081  036C  01
0082  036D  89                    .BYTE $89,$02,$9E,$01 ;      '4'
0082  036E  02
0082  036F  9E
0082  0370  01
0083  0371  89                    .BYTE $89,$02,$76,$01 ;      '5'
0083  0372  02
0083  0373  76
0083  0374  01
0084  0375  89                    .BYTE $89,$02,$53,$01 ;      '6'
0084  0376  02
0084  0377  53
0084  0378  01
0085  0379  4B                    .BYTE $4B,$02,$9E,$01 ;      '7'
0085  037A  02
0085  037B  9E
0085  037C  01
0086  037D  4B                    .BYTE $4B,$02,$76,$01 ;    '8'
0086  037E  02

LINE # LOC     CODE       LINE

0086  037F  76
0086  0380  01
0087  0381  4B                    .BYTE $4B,$02,$53,$01 ;         '9'
0087  0382  02
0087  0383  53
0087  0384  01
0088  0385                 .END

SYMBOL TABLE

SYMBOL   VALUE

ACR1     A00B   ACR2     A00B   DELAY    0355   DELCON   00FF
DIGIT    0302   NOEND    030A   NUMPTR   00C0   OFF      034C
OFFDEL   0020   ON       033C   ONDEL    0040   PHONE    0300
T1CH     A005   T1LH     A007   T1LL     A004   T2CH     AC05
T2LH     AC07   T2LL     AC04   TABLE    035D   WAIT     0357

END OF ASSEMBLY
```

Fig. 10-6 : Exemple de sortie de l'assembleur

Cela affecte à la variable ALPHA la valeur hexadécimale « A000 ». Les directives d'assemblage seront examinées dans une section ultérieure.

Constantes ou littéraux

On peut traditionnellement exprimer les constantes en décimal, en hexadécimal, en octal ou en binaire. Pour identifier la base utili-

sée pour représenter un nombre, on fait appel à un préfixe. Dans le cas décimal, on ne met pas de préfixe. Pour charger une constante dans l'accumulateur A, on écrira simplement :

LDA # 18 (où # dénote un littéral)

Un nombre hexadécimal sera précédé par $

Un symbole octal sera précédé par @

Un symbole binaire sera précédé par %

Par exemple, pour charger la valeur 11111111 dans l'accumulateur, on écrira :

LDA # % 11111111

On peut aussi utiliser des caractères ASCII dans une zone littérale. Dans les anciens assembleurs, on mettait traditionnellement le symbole ASCII entre apostrophes. Dans les assembleurs plus récents, pour avoir moins de caractères à taper on indique le type alphanumérique par une seule apostrophe précédant le symbole.

Par exemple, pour charger le code ASCII de la lettre « S » dans l'accumulateur on écrira :

LDA #'S

Pour charger l'apostrophe elle-même, on écrit, par convention :

LDA #'''

Exercice 10.1 : *Les deux instructions suivantes chargent-elles la même valeur dans l'accumulateur : LDA # '5 et LDA #$5 ?*

Opérateurs

Pour faciliter encore plus l'écriture des programmes symboliques, les assembleurs permettent d'utiliser des opérateurs. Au minimum il serait nécessaire d'autoriser « plus » et « moins », de façon, à pouvoir écrire, par exemple :

LDA ADR1 et :

LDX ADR1 + 1

Il est important de comprendre que l'expression ADR1 + 1 sera calculée par l'assembleur pour déterminer quelle est l'adresse véritable qui doit être insérée sous forme binaire. Elle sera calculée au *moment de l'assemblage,* et non lors de l'exécution du programme.

On peut avoir d'autres opérateurs comme la multiplication et la division, ce qui est commode pour accéder à des tables en mémoire. On peut aussi avoir des opérateurs plus spécialisés, comme par exemple « < » et « > », qui extraient respectivement l'octet haut et l'octet bas d'une valeur qui tient sur deux octets. Naturellement, le résultat d'une expression doit être positif. Les nombres négatifs ne

peuvent généralement pas être utilisés et ils doivent être exprimés en hexadécimal.

Enfin on utilise traditionnellement un symbole spécial pour représenter la valeur courante de l'adresse de la ligne : « * ». Ce symbole doit être interprété comme « adresse courante » (valeur de PC).

Exercice 10.2 : *Quelle est la différence entre les deux instructions suivantes ?*

LDA %10101010
LDA #%10101010

Exercice 10.3 : *Quel est l'effet de l'instruction BMI * — 2 ?*

Directives d'assemblage

Les directives sont des ordres spéciaux que le programmeur donne à l'assembleur, pour donner des valeurs aux symboles ou en introduire en mémoire, ou pour contrôler le mode de fonctionnement ou les impressions de l'assembleur. Pour donner un exemple réel, nous passons en revue les neuf directives disponibles sur le système de développement de Rockwell (« System 65 »). Ce sont :

Directive d'équivalence

On utilise un signe « = » pour affecter une valeur numérique à un symbole. Par exemple :

BASE = $1111
* = $1234

L'effet de la première directive est d'affecter à « BASE » la valeur « 1111 » en hexadécimal.

L'effet de la seconde est de forcer l'adresse de la ligne à être « 1234 » en hexadécimal. En d'autres termes, la prochaine instruction exécutable qu'on rencontrera sera implantée à l'adresse-mémoire « 1234 ».

Exercice 10.4 : *Ecrire une directive qui fait résider le programme à partir de l'adresse 0.*

Directives pour initialiser la mémoire

On dispose pour cela de trois directives : .BYT, .WOR, .DBY.

.BYT affecte les caractères ou les valeurs qui suivent, à des octets consécutifs en mémoire.

Exemple : RESERV .BYT 'SYBEX'

aura pour résultat de ranger les lettres « SYBEX » dans des emplacements successifs de la mémoire.

.WOR est utilisée pour ranger en mémoire des adresses sur deux octets, l'octet bas en premier.

Exemple : .WOR $1234,$2345

.DBY est identique à .WOR, sauf qu'elle range une valeur 16 bits, l'octet haut en premier. Elle est utilisée pour des données 16 bits plutôt que pour des adresses.

Les trois directives suivantes servent à contrôler les entrées-sorties.

Directives d'entrées-sorties

Ce sont .PAGE .SKIP .OPT

.PAGE commande à l'assembleur de terminer la page, c'est-à-dire d'aller en haut de la page suivante. On peut en outre spécifier un titre pour la page :

Exemple : .PAGE « titre de la page »

.SKIP sert à insérer des lignes blanches dans le listing. On peut spécifier le nombre de lignes à sauter.

Par exemple : .SKIP 3

.OPT spécifie quatre options : list, generate, error, symbol.

list fait générer une liste ;

generate sert à faire imprimer du code objet pour les chaînes de caractères données dans une autre directive .BYT ;

error indique si les diagnostics d'erreur doivent être imprimés ;

symbol fait imprimer la table des symboles.

Les deux dernières directives .FILE et .END contrôlent le format du listing d'assemblage.

Lorsqu'on développe un gros programme, on en écrit généralement plusieurs portions, mises au point séparément. A un certain moment, il est nécessaire de les assembler ensemble à partir de leurs fichiers. La dernière instruction du premier fichier contiendra la directive .FILE NOM/1 où 1 et le numéro de l'unité de disques et NOM le nom du prochain fichier. Le prochain fichier peut à son tour être lié à d'autres fichiers. A la fin du dernier fichier il y aura la directive .END NOM/1 qui pointe vers le premier fichier.

Enfin il y a un moyen pour insérer des commentaires supplémentaires dans le listing : un « ; » permet d'entrer à volonté des commentaires sur une ligne au lieu d'y mettre une instruction. C'est une possibilité intéressante si l'on veut que les programmes soient correctement documentés.

MACROS

Il n'y a actuellement pas de possibilités de macros sur la plupart des assembleurs qui existent pour le 6502. Néanmoins nous allons définir ce qu'est une macro et quels en sont les avantages. On peut espérer qu'il y aura bientôt des possibilités de macros sur la plupart des assembleurs pour 6502.

Une macro n'est autre qu'un nom attribué à un groupe d'instructions. C'est essentiellement une commodité pour le programmeur. Par exemple, si un groupe de cinq instructions est utilisé plusieurs fois dans un programme, on peut définir une macro plutôt que d'avoir à réécrire ces cinq instructions à chaque fois. Par exemple, on pourrait écrire :

```
SAVREG  MACRO
        PHA
        TXA
        PHA
        TYA
        PHA
        ENDM
```

et alors écrire le nom SAVREG au lieu des instructions ci-dessus.

A chaque fois qu'on écrit SAVREG, les cinq lignes correspondantes sont substituées à la place du nom. Un assembleur qui possède la facilité des macros est appelé macro-assembleur. Quand le macro-assembleur rencontre SAVREG il effectue une pure substitution physique des lignes correspondantes.

Macro ou sous-programme ?

A ce niveau, une macro semble fonctionner de façon analogue à un sous-programme. Ce n'est pas le cas. Quand on fait appel à l'assembleur pour produire le code objet, à chaque fois qu'on rencontre un nom de macro, il est remplacé par les instructions réelles qu'il représente. Au moment de l'exécution, le groupe d'instructions apparaît autant de fois que le nom de la macro apparaissait. Au contraire, un sous-programme est défini une seule fois et on peut l'utiliser de façon répétitive : le programme saute à l'adresse du sous-programme. La macro est une facilité qui joue au moment de l'assemblage. Le sous-programme joue au moment de l'exécution. Leur fonctionnement est très différent.

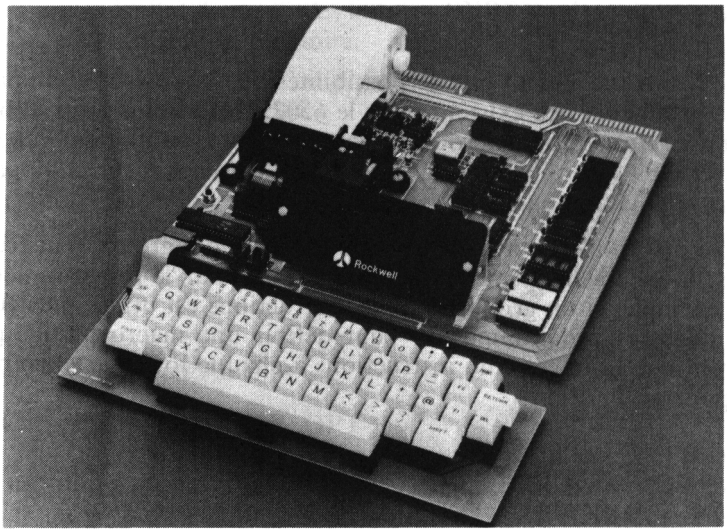


Fig. 10-7 : L'AIM-65 est une carte avec mini-imprimante et clavier complet

Fig. 10-8 : L'Ohio Scientific est un microordinateur pour amateurs

Paramètres des macros

Toute macro peut posséder un certain nombre de paramètres. Par exemple, considérons la macro suivante :

```
ECHNG   MACRO   M,N,T
        LDA     M
        STA     T
        LDA     N
        STA     M
        LDA     T
        STA     N
        ENDM
```

Cette macro a pour effet d'échanger les contenus des emplacements-mémoire M et N. Il n'existe pas dans le jeu d'instructions du 6502 d'opération d'échange de deux registres ou de deux cases-mémoire. On peut utiliser une macro pour la réaliser. T est simplement, dans cet exemple le nom d'un emplacement de stockage temporaire nécessaire au programme. Par exemple, échangeons les contenus des mémoires ALPHA et BETA. L'instruction qui le fait est :

```
ECHNG   ALPHA, BETA, TEMP
```

Dans cette instruction, TEMP est le nom d'une case-mémoire temporaire dont nous savons qu'elle est libre et peut être utilisée par la macro. Le résultat du développement de la macro apparaît ci-desssous :

```
LDA   ALPHA
STA   TEMP
LDA   BETA
STA   ALPHA
LDA   TEMP
STA   BETA
```

L'intérêt des macros devrait maintenant être patent : il est commode pour le programmeur d'utiliser des pseudo-instructions qui ont été définies à l'aide de macros.

De cette manière le jeu d'instructions apparent du 6502 peut être étendu à volonté. Malheureusement, il faut se souvenir que toute macro va se développer en autant d'instructions qu'il y en avait dans sa définition. Une macro sera donc plus lente que n'importe quelle

instruction toute seule. En raison de sa commodité pour le développement des longs programmes, la possibilité de macros est hautement désirable pour de telles applications.

Autres facilités des macros

On peut ajouter beaucoup d'autres possibilités aux macros simples : les macros peuvent être *emboîtées,* c'est-à-dire qu'un appel à une macro peut apparaître à l'intérieur d'une définition de macro. Avec cette possibilité, une macro peut se modifier elle-même dans une définition imbriquée ! Un premier appel produira un développement tandis que les appels suivants produiront un développement différent de la même macro.

ASSEMBLAGE CONDITIONNEL

L'assemblage conditionnel est une autre facilité des assembleurs qui n'a pas, jusqu'à maintenant été fournie sur la plupart des assembleurs pour 6502. L'assemblage conditionnel permet au programmeur d'utiliser les instructions spéciales « IF » suivie d'une expression, puis (éventuellement) « ELSE » et, à la fin, « ENDIF ». Lorsque l'expression qui suit IF est vraie, alors on assemblera les instructions présentes entre « IF » et « ELSE » s'il y a un ELSE ou « IF » et « ENDIF » s'il n'y a pas de ELSE.

Dans le cas où IF est suivi de ELSE, on assemble l'un ou l'autre de deux blocs d'instructions selon la valeur de l'expression testée.

Avec la possibilité d'assemblage conditionnel, le programmeur peut préparer un programme pour un ensemble de cas, puis assembler conditionnellement les segments nécessaires à une certaine application. Par exemple, un industriel peut concevoir des programmes capables de s'occuper de n'importe quel nombre de feux à une intersection, pour tout un ensemble d'algorithmes de commande. Il va alors recevoir les spécifications finales de la part du contrôleur de trafic, qui va indiquer combien il doit y avoir de feux et quels algorithmes on doit utiliser. Le programmeur n'aura plus qu'à définir quelques paramètres dans son programme et faire l'assemblage conditionnel, il en résultera un programme « adapté au client » qui ne retiendra que les routines nécessaires à la solution du problème.

L'assemblage conditionnel est donc d'un intérêt particulier pour la génération de programmes industriels dans un contexte où il y a beaucoup d'options et où le programmeur souhaite assembler rapidement et automatiquement des portions de programmes en fonctions de certains paramètres extérieurs.

RECAPITULATION

Ce chapitre a présenté les techniques et les outils hardware et software nécessaires pour développer un programme, ainsi que les différentes alternatives et les compromis qui en résultent.

Ils s'étendent, au niveau hardware, du microordinateur sur une carte au système de développement complet ; au niveau software, du codage binaire à la programmation en langage évolué.

Vous devrez effectuer un choix en fonction de vos buts et de vos moyens.

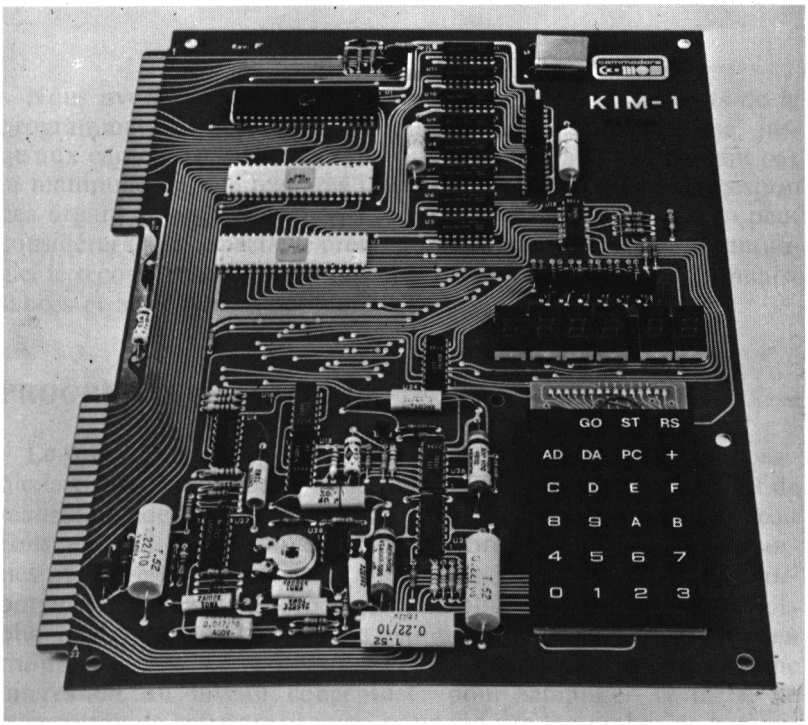


Fig. 10-9 : Le KIM 1 est une carte de coût minimal

11

CONCLUSION

Nous avons maintenant traité tous les aspects importants de la programmation, depuis les définitions et concepts de base, jusqu'aux caractéristiques des aides au développement, en passant par la manipulation des registres internes du 6502 ainsi que la gestion des organes d'entrées-sorties. Quelle est l'étape suivante ? On peut considérer deux aspects, le premier relatif au progrès de la technologie, le second concernant le développement de vos propres connaissances et aptitudes. Examinons-les :

PROGRES TECHNOLOGIQUE

Le progrès de l'intégration dans la technologie MOS rend possible la réalisation de boîtiers de plus en plus complexes. Le coût de réalisation de la fonction processeur proprement dite décroît constamment. Il en résulte que de nombreux boîtiers d'entrées-sorties ou de gestion de périphériques, utilisés dans un système, incorporent maintenant un processeur simplifié. Cela veut dire que la plupart des boîtiers LSI utilisés maintenant dans un système sont en train de devenir *programmables.* Il apparaît maintenant un dilemme intéressant au niveau conceptuel : pour simplifier la tâche de conception de software et aussi pour réduire le nombre de composants, les nouveaux boîtiers d'entrées-sorties ont maintenant des possibilités de programmation élaborées : de nombreux algorithmes programmés sont maintenant intégrés dans le boîtier. Néanmoins, il en résulte que le développement de programmes est compliqué par le fait que tous ces boîtiers d'entrées-sorties sont très différents et qu'ils doivent être étudiés en détail par le programmeur ! *Programmer un système n'est plus seulement programmer le microprocesseur ; c'est aussi programmer les différents autres boîtiers qui lui*

Fig. 11-1 : Le C.B.M. est un système de gestion complet avec disquette et imprimante

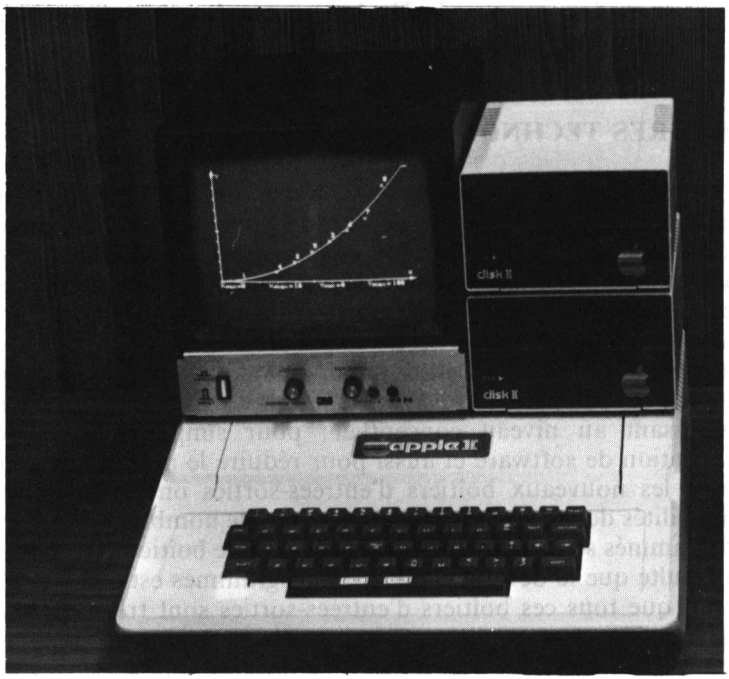


Fig. 11-2 : L'APPLE II utilise un téléviseur ordinaire

sont reliés. Le temps d'apprentissage pour chacun de ces boîtiers peut être considérable.

Naturellement le dilemme n'est qu'apparent. Si ces boîtiers n'étaient pas disponibles, la complexité de l'interface à réaliser et celle des programmes correspondants seraient encore plus grandes. La complexité nouvelle qui s'ajoute est qu'on a à programmer plus qu'un processeur pur et simple, et qu'il faut apprendre les caractéristiques diverses des différents boîtiers du système pour les utiliser efficacement. Cependant, on espère que les techniques et les concepts présentés dans ce livre, et le volume suivant, devraient en faire une tâche raisonnablement aisée.

LA PROCHAINE ETAPE

Vous avez maintenant appris les techniques de base, nécessaires pour programmer — sur le papier — des applications simples. C'était le but de ce livre. La prochaine étape est la pratique réelle. Il n'y a pas de substitut possible. Il est impossible d'apprendre la programmation uniquement sur le papier et il faut expérimenter. Vous devriez maintenant être en mesure de commencer à écrire vos propres programmes. On espère que vous y prendrez goût.

Pour ceux qui estiment qu'ils auraient avantage à être guidés par un autre livre, le volume complémentaire de celui-ci dans la série est le « livre des applications du 6502 » qui présente un ensemble d'applications concrètes pouvant tourner sur un microordinateur réel. C'est le livre des entrées-sorties (il contient également un assembleur complet pour le 6502, en BASIC).

Enfin, pour ceux qui souhaitent développer leur aptitude à la réalisation d'algorithmes complexes, le volume suivant est celui des « Jeux pour le 6502 ».

APPENDICE A

TABLE DE CONVERSION HEXADECIMALE

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

5		4		3		2		1		0	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393.216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

APPENDICE B

LISTE ALPHABETIQUE DES INSTRUCTIONS DU 6502

ADC	Addition avec retenue
AND	ET logique
ASL	Décalage arithmétique gauche
BCC	Branchement si C = 0
BCS	Branchement si C = 1
BEQ	Branchement si résultat = 0
BIT	Test de bits
BMI	Branchement si négatif
BNE	Branchement si non égal à 0
BPL	Branchement si positif ou nul
BRK	Interruption simulée
BVC	Branchement si non débordement
BVS	Branchement si débordement
CLC	Annuler retenue
CLD	Annuler le mode décimal
CLI	Annuler inhibition des interruptions
CLV	Annuler débordement
CMP	Comparer à l'accumulateur
CPX	Comparer à X
CPY	Comparer à Y
DEC	Décrémenter mémoire
DEX	Décrémenter X
DEY	Décrémenter Y
EOR	OU exclusif
INC	Incrémenter mémoire
INX	Incrémenter X
INY	Incrémenter Y
JMP	Saut inconditionnel
JSR	Appel de sous-programme
LDA	Charger accumulateur
LDX	Charger X
LDY	Charger Y
LSR	Décalage logique à droite
NOP	Pas d'opération
ORA	OU logique
PHA	Empiler A
PHP	Empiler P
PLA	Dépiler A
PLP	Dépiler P
ROL	Rotation à gauche
ROR	Rotation à droite
RTI	Retour d'interruption
RTS	Retour de sous-programme
SBC	Soustraction avec retenue
SEC	Mettre retenue à 1
SED	Mettre en mode décimal
SEI	Inhiber les interruptions
STA	Ranger l'accumulateur
STX	Ranger X
STY	Ranger Y
TAX	Transférer A dans X
TAY	Transférer A dans Y
TSX	Transférer S dans X
TXA	Transférer X dans A
TXS	Transférer X dans S
TYA	Transférer Y dans A

APPENDICE C

INSTRUCTIONS DU 6502 : HEXA ET TEMPS

		IMPLIED			ACCUM.			ABSOLUTE			ZERO PAGE			IMMEDIATE			ABS. X		
MNEMONIC		OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#
ADC	(1)							6D	4	3	65	3	2	69	2	2	7D	4	3
AND	(1)							2D	4	3	25	3	2	29	2	2	3D	4	3
ASL					OA	2	1	OE	6	3	06	5	2				1E	7	3
BCC	(2)																		
BCS	(2)																		
BEQ	(2)																		
BIT								2C	4	3	24	3	2						
BMI	(2)																		
BNE	(2)																		
BPL	(2)																		
BRK		00	7	1															
BVC	(2)																		
BVS	(2)																		
CLC		18	2	1															
CLD		D8	2	1															
CLI		58	2	1															
CLV		B8	2	1															
CMP								CD	4	3	C5	3	2	C9	2	2	DD	4	3
CPX								EC	4	3	E4	3	2	E0	2	2			
CPY								CC	4	3	C4	3	2	C0	2	2			
DEC								CE	6	3	C6	5	2				DE	7	3
DEX		CA	2	1															
DEY		88	2	1															
EOR	(1)							4D	4	3	45	3	2	49	2	2	5D	4	3
INC								EE	6	3	E6	5	2				FE	7	3
INX		E8	2	1															
INY		C8	2	1															
JMP								4C	3	3									
JSR								20	6	3									
LDA	(1)							AD	4	3	A5	3	2	A9	2	2	BD	4	3
LDX	(1)							AE	4	3	A6	3	2	A2	2	2			
LDY	(1)							AC	4	3	A4	3	2	A0	2	2	BC	4	3
LSR					4A	2	1	4E	6	3	46	5	2				5E	7	3
NOP		EA	2	1															
ORA								0D	4	3	05	3	2	09	2	2	1D	4	3
PHA		48	3	1															
PHP		08	3	1															
PLA		68	4	1															
PLP		28	4	1															
ROL					2A	2	1	2E	6	3	26	5	2				3E	7	3
ROR					6A	2	1	6E	6	3	66	5	2				7E	7	3
RTI		40	6	1															
RTS		60	6	1															
SBC	(1)							ED	4	3	E5	3	2	E9	2	2	FD	4	3
SEC		38	2	1															
SED		F8	2	1															
SEI		78	2	1															
STA								8D	4	3	85	2					9D	5	3
STX								8E	4	3	86	2							
STY								8C	4	3	84	2							
TAX		AA	2	1															
TAY		A8	2	1															
TSX		BA	2	1															
TXA		8A	2	1															
TXS		9A	2	1															
TYA		98	2	1															

(1) *Add 1 to n if crossing page boundary*

(2) Add 2 to n if branch within page
Add 3 to n if branch to another page

ABS. Y			(IND. X)			(IND)Y			Z. PAGE, X			RELATIVE			INDIRECT			Z. PAGE, Y			PROCESSOR STATUS CODES
OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	N V B D I Z C
79	4	3	61	6	2	71	5	2	75	4	2										● ● ● ●
39	4	3	21	6	2	31	5	2	35	4	2										● ●
									16	6	2										● ● ●
												90	2	2							
												BO	2	2							
												FO	2	2							M7 M6 ●
												30	2	2							
												DO	2	2							
												10	2	2							
												50	2	2							1 1
												70	2	2							0
																					0
																					0
D9	4	3	C1	6	2	D1	5	2	D5	4	2										0 ● ● ● ● ●
									D6	6	2										● ● ● ● ●
59	4	3	41	6	2	51	5	2	55	4	2										●
									F6	6	2										●
B9	4	3	A1	6	2	B1	5	2	B5	4	2				6C	5	3				● ● ●
BE	4	3																B6	4	2	● ●
									B4	4	2										● ●
									56	6	2										0 ● ●
19	4	3	01	6	2	11	5	2	15	4	2										● ●
																					●
									36	6	2										● ● ● ● ● ● ● ● ●
									76	6	2										● ●
F9	4	3	E1	6	2	F1	5	2	F5	4	2										● ● ● ● ● ● ● ● ●

APPENDICE D

LISTE DES CODES BINAIRES DES INSTRUCTIONS DU 6502

ADC	011bbb01	LDX	101bbb10
AND	001bbb01	LDY	101bbb00
ASL	000bbb10	LSR	010bbb10
BCC	10110000	NOP	11101010
BEQ	11110000	ORA	000bbb01
BIT	0010b100	PHP	01001000
BMI	00110000	PHP	00001000
BNE	11010000	PLA	01101000
BPL	00010000	PLP	00101000
BRK	01010000	ROL	001bbb10
CLC	00011000	ROR	011bbb10
CLD	11011000	RTI	01000000
CLI	01011000	RTS	01100000
CMP	110bbb01	SBC	111bbb01
CPX	1110bb00	SEC	00111000
CPY	1100bb00	SED	11111000
DEC	110bb110	SEI	01111000
DEX	11001010	STA	100bbb01
DEY	10001000	STX	100bb110
EOR	101bbb01	STY	100bb100
INC	111bb110	TAX	10101010
INX	11101000	TAY	10101000
INY	11001000	TSX	10111010
JMP	01b01100	TXA	10001010
JSR	00100000	TXS	10011010
LDA	101bbb01	TYA	10011000

APPENDICE E

TABLE DE CONVERSION ASCII

HEX	BITS	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB		7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

SYMBOLES ASCII SPECIAUX

NUL —Null
SOH —Start of Heading
STX —Start of Text
ETX —End of Text
EOT —End of Transmission
ENQ —Enquiry
ACK —Acknowledge
BEL —Bell
BS —Backspace
HT —Horizontal Tabulation
LF —Line Feed
VT —Vertical Tabulation
FF —Form Feed
CR —Carriage Return
SO —Shift Out
SI —Shift In

DLE —Data Link Escape
DC —Device Control
NAK —Negative Acknowledge
SYN —Synchronous Idle
ETB —End of Transmission Block
CAN —Cancel
EM —End of Medium
SUB —Substitute
ESC —Escape
FS —File Separator
GS —Group Separator
RS —Record Separator
US —Unit Separator
SP —Space (Blank)
DEL —Delete

APPENDICE F

TABLES DE BRANCHEMENTS RELATIFS

BRANCHEMENTS RELATIFS EN AVANT

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

BRANCHEMENTS RELATIFS EN ARRIERE

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

APPENDICE G

LISTE PAR CODE OPERATION HEXADECIMAL

LSD / MSD	0	1	2	3	4	5	6	7
0	BRK	ORA-I, X				ORA-Ø-P	ASL-Ø-P	
1	BPL	ORA-I, Y				ORA-Ø-P, X	ASL-Ø-P, X	
2	JSR	AND-I, X			BIT-Ø-P	AND-Ø-P	ROL-Ø-P	
3	BMI	AND-I, Y				AND-Ø-P, X	ROL-Ø-P, X	
4	RTI	EOR-I, X				EOR-Ø-P	LSR-Ø-P	
5	BVC	EOR-I, Y				EOR-Ø-P, X	LSR-Ø-P, X	
6	RTS	ADC-I, X				ADC-Ø-P	ROR-Ø-P	
7	BVS	ADC-I, Y				ADC-Ø-P, X		
8		STA-I, X			STY-Ø-P	STA-Ø-P	STX-Ø-P	
9	BCC	STA-I, Y			STY-Ø-P, X	STA-Ø-P, X	STX-Ø-P, Y	
A	LDY-IMM	LDA-I, X	LDX-IMM		LDY-Ø-P	LDA-Ø-P	LDX-Ø-P	
B	BCS	LDA-I, Y			LDY-Ø-P, X	LDA-Ø-P, X	LDX-Ø-P, Y	
C	CPY-IMM	CMP-I, X			CPY-Ø-P	CMP-Ø-P	DEC-Ø-P	
D	BNE	CMP-I, Y				CMP-Ø-P, X	DEC-Ø-P, X	
E	CPX-IMM	SBC-I, X			CPX-Ø-P	SBC-Ø-P	INC-Ø-P	
F	BEQ	SBC-I, Y				SBC-Ø-P, X	INC-Ø-P, X	

8	9	A	B	C	D	E	F	LSD / MSD
PHP	ORA-IMM	ASL-A			ORA	ASL		0
CLC	ORA, Y				ORA, X	ASL, X		1
PLP	AND-IMM	ROL-A		BIT	AND	ROL		2
SEC	AND, Y				AND, X	ROL, X		3
PHA	EOR-IMM	LSR-A		JMP	EOR	LSR		4
CLI	EOR, Y				EOR, X	LSR, X		5
PLA	ADC-IMM	ROR-A		JMP-I	ADC	ROR		6
SEI	ADC, Y				ADC, X			7
DEY		TXA		STY	STA	STX		8
TYA	STA, Y	TXS			STA, X			9
TAY	LDA-IMM	TAX		LDY	LDA	LDX		A
CLV	LDA, Y	TSX		LDY, X	LDA, X	LDX, Y		B
INY	CMP-IMM	DEX		CPY	CMP	DEC		C
CLD	CMP, Y				CMP, X	DEC, X		D
INX	SBC-IMM	NOP		CPX	SBC	INC		E
SED	SBC, Y				SBC, X	INC, X		F

I = indirect

Ø-P = page zero

INDEX

A

absolu 183
activation par niveau 232
accès à un bloc séquentiel 192
accumulateur 38
ADC 106
addition 50
addition DCB 59
adressage 181
adressage absolu 5, 188
adressage direct 77, 184
adressage immédiat 61, 183, 187
adressage implicite 183, 187
adressage indexé 184, 189, 227
adressage indexé indirect 185, 191
adressage indirect 186, 190
adressage indirect indexé 190
adressage page-zéro 188
adressage physique 70
adressage relatif 184, 188
adresse 36
adresse courte 187, 192
adresse de vectorisation 234
aides software 330
ajustement décimal 60
algorithme 7
alternatives de programmation 75
alphanumérique 28
amplificateurs 38
amplifie 38
AND 98, 108
annuler 104
appel 96
appel de sous-programme 85
appels impriqués 86
arbres 269
architecture de système 35
arithmétique 94
ASCII 29
ASL 110
assemblage conditionnel 360
assembleur 51, 327, 330, 340
assembleurs croisés 338
asynchrone 212

B

bascule 39
BASIC 329
baud 225
BCC 112
BCS 113
benchmark 211
BEQ 114
bibliothèque de sous-programmes 91
binaire 344
binaire direct 11
binaire signée 15
BIT 103, 115
bit 10
bit de départ 223, 224
bloc 264
BMI 116
BNE 117
boîtiers combinés 38
boîtiers d'interface 38
boucles 64, 184
boucle de programme 64
BPL 118
branches 188
branchement 95, 181
branchement à distance 192
break 96, 101, 240
BRK 228, 101, 119, 240
BUS 36
BUS adresse 36
BUS de commande 36
BUS de données 36
BVC 120
BVS 121

C

C 42
caractères 28
chaîne 220
champs 340
champ adresse 340
champ commentaire 51, 340
champ d'étiquette 340
champs d'instruction 340
champ operande 340
chargement 145
chargeur 331
chargeur initial 37
chargeur-éditeur de liens 331
chiffre binaire 10
classes d'instructions 93
clavier 34
CLC 53, 122

CLD 54, 123
CLI 124
CLV 125
CMP 126
code objet 272
code opération 39
codage 8, 332
code hexadécimal 327
combinaisons 187
comparaisons 100
compilateur 330
complément 99
concepts de base 7
complément à dix 61
complément à un 16
complément à deux 17
comptage 205
comptage d'impulsions 209
compteur 66
concepts hardware 35
conseils de programmation 67
constantes 343
contrôle de processus 37
contrôleur de priorité d'interruption 238
conversion de codes 256
CPX 104, 128
CPY 104, 130

D

D 54, 101
débordement 100
débordement de la pile 242
DEC 132
décalage 95, 151
DCB compacte 26, 59
décimal 24, 54, 101
décrémentation 94
délai 205
délais hardware 209
délais prolongés 206
dépannage 9
dépiler 44, 158
déplacement 103, 184
désassembleur 271
destination 36
détection d'impulsion 209
développement de programmes 327
dévermineur 331
DEX 134
DEY 135
dichotomique 270
directive 88, 328, 345
directives d'assemblage 345
dispositifs extérieurs 36
division avec rétablissement 80
division binaire 79
DMA 228
documentation 51
données 36
DOS 331
durée 208

E

EBCDIC 28
écho 224
éditeur 331
élément maximul 335
emboîte 350
émission d'un caractère 220
empiler 44, 156
emplacement 68
émulateur 332
émulateur sur circuit utilisateur 264, 334
entrées-sorties 96, 203
entrées-sorties projetées en mémoire 96
entrées-sorties sur télétype 223
EOR 99, 136
étiquette symbolique 31
exécuteur 331
exposant 27
extension du signe 94

F

fifo 267
file d'attente 267

G

génération de parité 255
génération d'un signal 204
gestion des registres 49
gestionnaire de périphérique 238, 332

H

hexadécimal 344
horloge 36, 42

I

I 232, 240
impression d'une chaîne de caractères 227
imprimantes 220, 227
impulsions 203, 204, 210
INC 138
incrémentation 94
indicateurs DCB 62
indicateurs d'état 95, 100
initialisation 65
instruction 10, 51

instructions de contrôle 96, 104
instructions d'entrées-sorties 105
instruction suivante 41, 42
interpréteur 330
interruption 44, 96, 102, 164, 232
interruptions multiples 238
interruption non masquable 232
interruptions simultanées 238
INX 140
INY 141
IR 41
IRQ 47, 232
itération 197

J

jeu d'instructions 93
JMP 103
JSR 88, 103, 143

K

K 15
KIM 250, 351

L

langage évolué 329
langage de niveau assembleur 328
langages de programmation 329
LDA 145
LDX 147
LDY 149
LED 220
lifo 44
lignes de commande 244
liste 266
liste circulaire 268
liste directe et réciproque 270
listes séquentielles 264
listing 88, 330, 340
littéral 60, 183, 344
logique 94
logique de décodage 38
LSR 151

M

macros 347
manipulation des indicateurs 62
mantisse 27
masquage 98
masque d'interruption 102, 232
mémoire 37
mémoire à lecture-écriture 37
mémoire à lecture seule 37
mémoire vive 37
messages d'erreur 342
mettre à un 104, 168, 169, 170
microordinateurs pour amateurs 329
microordinateurs sur une seule carte 328, 335
mnémoniques 328
modes d'adressage 181
mode binaire 101
modes DCB 101
mode décimal 169
moniteur 37, 331
monostable 207
MOS technology 250
MPU 35
multiplicande 71
multiplication 63
multiplication améliorée 76

N

N 39
négatif 39
niveaux d'interruption 239
niveaux de sous-programmes 88
niveaux 88
NMI 220, 47, 232
NOP 153
normalisation 27

O

octal 30, 344
octet 10
opérande 39
opérateurs 344
opération arithmétique 38
opérations de décalage 94, 95
opérations logiques 38, 98
opérations sur la pile 97
ORA 98, 154
organes d'entrées-sorties 243
organisation 228
organisation interne 38
ordinogrammes 8
ordre alphabétique 257
oscillateur 37
oscilloscope 334
ou inclusif 52

P

page 0 45
pagination 45
panneau de commande 339
pannes d'alimentation 37
paramètres 89
paramètres des macros 349

parité 29
particularités du 6502 53
PASCAL 329
PC 40
PCH 40
PCL 40
périphérique 36
PHA 156
phase de l'assemblage 344
PHP 157
PIC 238
pile 44, 86, 232, 263, 268
pile hardware 44
PIO 37, 243
PIO standard 243
PLA 158
PLP 159
point d'arrêt 240
point de branchement 84
pointeurs 185, 263
port 38, 243
ports d'entrée 38
post-indexation 185
précision 24
précision étendue 54
pré-indication 185
priorité 238
produit partiel 63
programmateur de prom 282
programme 37
programmes arithmétiques 50
programme principal 84
programme de test de performances 211
programmes utilitaires 251

Q

quartet 10
quartz 37

R

RAM 37
rangement 170, 182
RDY 47
recherche 41, 42
recherche et classement 270
recherche dichotomique 270
recherche logarithmique 270
recherche séquentielle 270
récupération 268
récursivité 88
registre 38, 68
registre de commande 248
registre direction 248
registre d'état 232
registres d'index 43, 184
registre instruction 41
registres de travail 188
relais 204
remise à zéro 245
répertoires 265
répertoire principal 265
représentation binaire 10
représentation des informations 10
RES 47
reset 245
retenue 40, 53, 102
retenue intermédiaire 60
retour 84, 164, 165
rockwell 250
ROL 160
ROM 37
ROR 162
rotation 67, 180, 162
routine de service d'interruption 232
routines utilitaires 278
RTI 103, 164, 236
RTS 88, 103, 165
RW 47

S

S 43
saut 180, 96, 142
saut 95
saut inconditionnel 95
SBC 166
scrutation 208, 129, 252
SEC 59, 168
SED 169
SEI 170
séquencement 36
seudo-instructions 54
signaux de commande 47
signe 100
simulateur 331
six-mille-cinq-cent-deux 35
SO 47
somme de contrôle 258
somme de n éléments 257
source 36
sous-programme 44, 84, 165
soustraction 58
soustraction DCB 61
soustraire 166
STA 171
structures de données 263
STX 173
STY 174
SYM 250
symbole 345
SYNC 47
synchrone 212
synchronisation 36, 96
synchronisation des échanges 218, 244

synertek 250
syntaxe 269
system 65 345
système de développement 336
système d'exploitation 301
système d'exploitation disque 301
système de fichiers 265
système en temps partagé 338

T

table 184
table des symboles 342
tampon 244
taux de transfert 212
TAX 175
TAY 176
techniques d'adressage 181
techniques de programmation 49
techniques de recherche 271
test dans un intervalle 254
temporisateur 207, 247
temps perdu 236
test 254
test et branchement 93, 95, 100
tests de mémoires 252
tracé 335
traduction 51
traitement de données 244
transfert 175
transfert de données 62, 72
transfert de mot en parallèle 210
transfert série 212
TRI 257
troncation 24
TSX 177
TXA 178
TXS 179
TYA 180
types d'instructions 62

U

unité arithmétique et logique 36
unité de commande 42

V

val 36
vecteur d'interruption 234
VIA 247
virgule flottante 27, 94
vitesse d'exécution 39
volatile 37

W

wart 218

X

X 43

Y

Y 43

Z

Z 39, 102
zéro 39, 102

LA BIBLIOTHÈQUE SYBEX

OUVRAGES GÉNÉRAUX

VOTRE PREMIER ORDINATEUR *par Rodnay Zaks,*
296 pages, Réf. 394

VOTRE ORDINATEUR ET VOUS *par Rodnay Zaks,*
296 pages, Réf. 271

DU COMPOSANT AU SYSTÈME : une introduction aux microprocesseurs *par Rodnay Zaks,*
636 pages, Réf. 0040

TECHNIQUES D'INTERFACE aux microprocesseurs *par Austin Lesea et Rodnay Zaks,*
450 pages, Réf. 0039

LEXIQUE INTERNATIONAL MICRO-ORDINATEURS, avec dictionnaire abrégé en 10 langues
192 pages, Réf. 234

GUIDE DES MICRO-ORDINATEURS A MOINS 3 000 F *par Joël Poncet,*
144 pages, Réf. 322

LEXIQUE MICRO-INFORMATIQUE *par Pierre Le Beux,*
140 pages, Réf. 369

LA SOLUTION RS-232 *par Joe Campbell,*
208 pages, Réf. 0052

MINITEL ET MICRO-ORDINATEUR *par Pierrick Bourgault,*
198 pages, Réf. 0119

ROBOTS - CONSTRUCTION, PROGRAMMATION *par Fernand Esteves,*
400 pages, Réf. 0130

ALGORITHMES *par Pierre Beaufils et Wolfram Luther,*
296 pages, Réf. 0149

BASIC

VOTRE PREMIER PROGRAMME BASIC *par Rodnay Zaks,*
208 pages, Réf. 263

INTRODUCTION AU BASIC *par Pierre Le Beux,*
336 pages, Réf. 0035

LE BASIC PAR LA PRATIQUE : 60 exercices *par Jean-Pierre Lamoitier,*
252 pages, Réf. 0095

LE BASIC POUR L'ENTREPRISE *par Xuan Tung Bui,*
204 pages, Réf. 253

PROGRAMMES EN BASIC, Mathématiques, Statistiques, Informatique *par Alan R. Miller,*
318 pages, Réf. 259

BASIC, PROGRAMMATION STRUCTURÉE *par Richard Mateosian,*
352 pages, Réf. 0129

JEUX D'ORDINATEUR EN BASIC *par David H. Ahl,*
192 pages, Réf. 246

NOUVEAUX JEUX D'ORDINATEUR EN BASIC *par David H. Ahl,*
204 pages, Réf. 247

FICHIERS EN BASIC *par Alan Simpson,*
256 pages, Réf. 0102

TECHNIQUES DE PROGRAMMATION EN BASIC *par S. Crosmarie, M. Perron et D. Philippine*
152 pages, Réf. 0124

PASCAL

INTRODUCTION AU PASCAL *par Pierre Le Beux,*
496 pages, Réf. 0030

LE PASCAL PAR LA PRATIQUE *par Pierre Le Beux et Henri Tavernier,*
562 pages, Réf. 361

LE GUIDE DU PASCAL *par Jacques Tiberghien,*
504 pages, Réf. 423

PROGRAMMES EN PASCAL pour Scientifiques et Ingénieurs *par Alan R. Miller,*
392 pages, Réf. 240

AUTRES LANGAGES

INTRODUCTION A ADA *par Pierre Le Beux,*
366 pages, Réf. 360

INTRODUCTION A C *par Bruce Hunter,*
312 pages, Réf. 0092

MICRO-ORDINATEURS

ALICE

JEUX EN BASIC POUR ALICE *par Pierre Monsaut,*
96 pages, Réf. 320

ALICE et ALICE 90, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 376

ALICE, 56 PROGRAMMES *par Stanley R. Trost,*
160 pages, Réf. 401

ALICE, GUIDE DE L'UTILISATEUR *par Norbert Rimoux,*
208 pages, Réf. 378

ALICE, PROGRAMMATION EN ASSEMBLEUR *par Georges Fagot-Barraly,*
192 pages, Réf. 420

AMSTRAD

AMSTRAD, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 0105

AMSTRAD, 56 PROGRAMMES *par Stanley R. Trost,*
160 pages, Réf. 0107

AMSTRAD, JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 0108

AMSTRAD, PROGRAMMATION EN ASSEMBLEUR
par Georges Fagot-Barraly,
208 pages, Réf. 0136

AMSTRAD EXPLORÉ *par John Braga,*
192 pages, Réf. 0135

AMSTRAD, GUIDE DU GRAPHISME *par James Wynford,*
208 pages, Réf. 0141

AMSTRAD CP/M 2.2 *par Anatole D'Hardencourt,*
248 pages, Réf. 0156

AMSTRAD ASTROLOGIE/NUMEROLOGIE/BIORYTHMES
par Pierrick Bourgault,
160 pages, Réf. 0167

AMSTRAD MULTIPLAN *de Microsoft,*
496 pages, Réf. 1111

AMSTRAD, CRÉER DE NOUVELLES INSTRUCTIONS
par Jean-Claude Despoine,
144 pages, Réf. 0176

AMSTRAD ASTROCALC
par Gérard Blanc et Philippe Destrebecq,
168 pages, Réf. 0162

APPLE / MACINTOSH

PROGRAMMEZ EN BASIC SUR APPLE II,
Tomes 1 et 2 *par Léopold Laurent,*
208 pages, Réf. 333 et 380

APPLE II 66 PROGRAMMES BASIC *par Stanley R. Trost,*
192 pages, Réf. 283

JEUX EN PASCAL SUR APPLE
par Douglas Hergert et Joseph T. Kalash,
372 pages, Réf. 241

GUIDE DU BASIC APPLE II *par Douglas Hergert,*
272 pages, Réf. 0006

APPLE II, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 373

MACINTOSH, GUIDE DE L'UTILISATEUR
par Joseph Caggiano,
208 pages, Réf. 396

APPLE IIC, GUIDE DE L'UTILISATEUR
par Thomas Blackadar,
160 pages, Réf. 0089

MULTIPLAN SUR MACINTOSH
par Goulven Habasque,
240 pages, Réf. 0099

INTRODUCTION A MAC PASCAL *par Pierre Le Beux,*
416 pages, Réf. 0145

MACINTOSH POUR LA PRESSE, L'EDITION ET
LA PUBLICITE *par Bernard Le Du,*
160 pages, Réf. 0173

ATARI

JEUX EN BASIC SUR ATARI *par Paul Bunn,*
96 pages, Réf. 282

ATARI, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 387

ATARI, GUIDE DE L'UTILISATEUR *par Thomas Blackadar,*
192 pages, Réf. 354

ATMOS

JEUX EN BASIC SUR ATMOS *par Pierre Monsaut,*
96 pages, Réf. 346

ATMOS, 56 PROGRAMMES *par Stanley R. Trost,*
180 pages, Réf. 372

COMMODORE 64

JEUX EN BASIC SUR COMMODORE 64
par Pierre Monsaut,
96 pages, Réf. 0017

COMMODORE 64, PREMIERS PROGRAMMES
par Rodnay Zaks,
248 pages, Réf. 342

GUIDE DU BASIC VIC 20, COMMODORE 64
par Douglas Hergert,
240 pages, Réf. 312

COMMODORE 64, GUIDE DE L'UTILISATEUR
par J. Kascmer,
144 pages, Réf. 314

COMMODORE 64, 66 PROGRAMMES
par Stanley R. Trost,
192 pages, Réf. 319

COMMODORE 64, GUIDE DU GRAPHISME
par Charles Platt,
372 pages, Réf. 0053

COMMODORE 64, JEUX D'ACTION *par Eric Ravis,*
96 pages, Réf. 403

COMMODORE 64, 1ers CONTACTS
par Marty Dejonghe et Caroline Earhart,
208 pages, Réf. 390

COMMODORE 64, BASIC APPROFONDI
par Gary Lippman,
216 pages, Réf. 0100

DRAGON

JEUX EN BASIC SUR DRAGON *par Pierre Monsaut,*
96 pages, Réf. 324

EXL 100

EXL 100, JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 0126

GOUPIL

PROGRAMMEZ VOS JEUX SUR GOUPIL
par François Abella,
208 pages, Réf. 264

HECTOR

HECTOR JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 388

IBM

IBM PC EXERCICES EN BASIC *par Jean-Pierre Lamoitier,*
256 pages, Réf. 338

IBM PC GUIDE DE L'UTILISATEUR
par Joan Lasselle et Carol Ramsey,
160 pages, Réf. 301

IBM PC 66 PROGRAMMES BASIC *par Stanley R. Trost,*
192 pages, Réf. 359

GRAPHIQUES SUR IBM PC *par Nelson Ford,*
320 pages, Réf. 357

GUIDE DE PC DOS *par Richard A. King,*
240 pages, Réf. 0013

LASER

LASER JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 371

MO 5

MO 5 JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 0067

MO 5, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 370

MO 5, 56 PROGRAMMES *par Stanley R. Trost,*
160 pages, Réf. 375

MO 5, PROGRAMMATION EN ASSEMBLEUR
par Georges Fagot-Barraly,
192 pages, Réf. 384

MO 5, DYNAMIQUE CINÉMATIQUE, MÉTHODE POUR LA PROGRAMMATION DES JEUX *par Daniel Lebigre,*
272 pages, Réf. 0118

MO 5, STATIQUE, DYNAMIQUE, ELECTRONIQUE, PROGRAMMES DE PHYSIQUE EN BASIC
par Beaufils, Lamarche et Muggianu,
240 pages, Réf. 0148

MO 5, PROGRAMMES D'ELECTRONIQUE EN BASIC
par Beaufils, Delusurieux, Do, Romanacce,
312 pages, Réf. 0143

MO 5, OPTIQUE, THERMODYNAMIQUE, CHIMIE
par P. Beaufils, M. Lamarche, Y. Muggianu,
224 pages, Réf. 0161

MSX

MSX, JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 411

MSX, INITIATION AU BASIC *par Rodnay Zaks,*
248 pages, Réf. 410

MSX, 56 PROGRAMMES *par Stanley R. Trost,*
160 pages, Réf. 0109

MSX, GUIDE DU GRAPHISME *par Mike Shaw,*
192 pages, Réf. 0132

MSX, PROGRAMMES EN LANGAGE MACHINE
par Steeve Webb,
112 pages, Réf. 0153

MSX, PROGRAMMATION EN ASSEMBLEUR
par Georges Fagot-Barraly,
216 pages, Réf. 0144

MSX, GUIDE DU BASIC *par Michel Laurent,*
264 pages, Réf. 0155

MSX, JEUX EN ASSEMBLEUR *par Eric Ravis*
112 pages, Réf. 0170

MSX, ROUTINES GRAPHIQUES EN ASSEMBLEUR
par Steeve Webb
88 pages, Réf. 0154

MSX, TECHNIQUES DE PROGRAMMATION DES JEUX EN ASSEMBLEUR
par Georges Fagot-Barraly,
176 pages, Réf. 0178

MSX ASTROLOGIE/NUMEROLOGIE/BIORYTHMES
par Pierrick Bourgault,
157 pages, Réf. 0168

ORIC

JEUX EN BASIC SUR ORIC *par Peter Shaw,*
96 pages, Réf. 278

ORIC PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 344

SHARP

DÉCOUVREZ LE SHARP PC-1500 ET LE TRS-80 PC-2
par Michel Lhoir,
2 tomes, Réf. 261-262

SPECTRAVIDEO

SPECTRAVIDEO, JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 377

SPECTRUM

PROGRAMMEZ EN BASIC SUR SPECTRUM
par S.M. Gee,
208 pages, Réf. 252

JEUX EN BASIC SUR SPECTRUM *par Peter Shaw,*
96 pages, Réf. 276

SPECTRUM, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 381

SPECTRUM JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 368

TI 99/4

PROGRAMMEZ VOS JEUX SUR TI 99/4
par François Abella,
160 pages, Réf. 303

TO 7

JEUX EN BASIC SUR TO 7 *par Pierre Monsaut,*
96 pages, Réf. 0026

TO 7, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 328

TO 7, PROGRAMMATION EN ASSEMBLEUR
par Georges Fagot-Barraly,
192 pages, Réf. 350

JEUX SUR TO 7 et MO 5 *par Georges Fagot-Barraly,*
168 pages, Réf. 0134

GESTION DE FICHIERS SUR TO 7 ET MO 5
par Jean-Pierre Lhoir,
136 pages, Réf. 0127

TO 7, 56 PROGRAMMES *par Stanley R. Trost,*
160 pages, Réf. 374

TO 7 / MO 5, GUIDE DU BASIC
par Jean-Louis Greco et Michel Laurent,
288 pages, Réf. 0158

TO 7 / MO 5, GUIDE DU GRAPHISME
par Michel Lamarche et Yves Muggianu,
240 pages, Réf. 0172

TO 7 / MO 5 ASTROLOGIE/NUMEROLOGIE/BIORYTHMES
par Pierrick Bourgault,
160 pages, Réf. 0169

TRS-80

PROGRAMMEZ EN BASIC SUR TRS-80
par Léopold Laurent,
2 tomes, Réf. 366-251

JEUX EN BASIC SUR TRS-80 MC-10 *par Pierre Monsaut,*
96 pages, Réf. 323

JEUX EN BASIC SUR TRS-80 *par Chris Palmer,*
96 pages, Réf. 302

JEUX EN BASIC SUR TRS-80 COULEUR
par Pierre Monsaut,
96 pages, Réf. 325

TRS-80 MODÈLE 100, GUIDE DE L'UTILISATEUR
par Orson Kellog,
112 pages, Réf. 300

TRS-80 COULEUR, PREMIERS PROGRAMMES
par Rodnay Zaks,
248 pages, Réf. 414

TRS-80 COULEUR, 56 PROGRAMMES
par Stanley R. Trost,
160 pages, Réf. 413

VIC 20

PROGRAMMEZ EN BASIC SUR VIC 20
par G. O. Hamann,
2 tomes, Réf. 329-337

JEUX EN BASIC SUR VIC 20 *par Alastair Gourlay,*
96 pages, Réf. 277

VIC 20, PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 341

VIC 20 JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 345

VG 5000

VG 5000, JEUX D'ACTION *par Pierre Monsaut,*
96 pages, Réf. 422

VG 5000, 56 PROGRAMMES *par Stanley R. Trost,*
160 pages, Réf. 0128

ZX 81

ZX 81 GUIDE DE L'UTILISATEUR *par Douglas Hergert,*
208 pages, Réf. 351

ZX 81 56 PROGRAMMES BASIC *par Stanley R. Trost,*
192 pages, Réf. 304

GUIDE DU BASIC ZX 81 *par Douglas Hergert,*
204 pages, Réf. 285

JEUX EN BASIC SUR ZX 81 *par Mark Charlton,*
96 pages, Réf. 275

ZX 81 PREMIERS PROGRAMMES *par Rodnay Zaks,*
248 pages, Réf. 343

MICROPROCESSEURS

PROGRAMMATION DU Z80 *par Rodnay Zaks,*
618 pages, Réf. 0058

APPLICATIONS DU Z80 *par James W. Coffron,*
304 pages, Réf. 0181

PROGRAMMATION DU 6502 *par Rodnay Zaks,*
376 pages, Réf. 0031, 2ème édition

APPLICATIONS DU 6502 *par Rodnay Zaks,*
288 pages, Réf. 332

PROGRAMMATION DU 6800
par Daniel-Jean David et Rodnay Zaks,
374 pages, Réf. 327

PROGRAMMATION DU 6809
par Rodnay Zaks et William Labiak,
392 pages, Réf. 0139

PROGRAMMATION DU 8086/8088
par James W. Coffron,
304 pages, Réf. 0016

MISE EN OEUVRE DU 68000 *par C. Vieillefond,*
352 pages, Réf. 0133

ASSEMBLEUR DU 8086/8088 *par François Retoreau,*
616 pages, Réf. 0093

SYSTÈMES D'EXPLOITATION

GUIDE DU CP/M AVEC MP/M *par Rodnay Zaks,*
354 pages, Réf. 336

CP/M APPROFONDI *par Alan R. Miller,*
380 pages, Réf. 334

INTRODUCTION AU p-SYSTEM UCSD
par Charles W. Grant et Jon Butah,
308 pages, Réf. 365

GUIDE DE MS-DOS *par Richard A. King,*
360 pages, Réf. 0117

INTRODUCTION A UNIX *par John D. Halamka,*
240 pages, Réf. 0098

GUIDE DE PRODOS
par Pierre Beaufils et Wolfram Luther,
248 pages, Réf. 0146

APPLICATIONS ET LOGICIELS

INTRODUCTION AU TRAITEMENT DE TEXTE
par Hal Glatzer,
228 pages, Réf. 243

INTRODUCTION A WORDSTAR *par Arthur Naiman,*
200 pages, Réf. 0062

WORDSTAR APPLICATIONS *par Julie Anne Arca,*
320 pages, Réf. 0005

VISICALC APPLICATIONS *par Stanley R. Trost,*
304 pages, Réf. 258

VISICALC POUR L'ENTREPRISE *par Dominique Helle,*
304 pages, Réf. 309

INTRODUCTION A dBASE II *par Alan Simpson,*
280 pages, Réf. 0064

DE VISICALC A VISI ON *par Jacques Bourdeu,*
256 pages, Réf. 321

MULTIPLAN POUR L'ENTREPRISE
par D. Helle et G. Boussand,
304 pages, Réf. 0079

dBASE II APPLICATIONS *par Christophe Stehly,*
248 pages, Réf. 416

INTRODUCTION A LOTUS 1-2-3
par Chris Gilbert et Laurie Williams,
272 pages, Réf. 0106

INTRODUCTION A dBASE III *par Alan Simpson,*
272 pages, Réf. 0131

LOTUS 1-2-3 POUR L'ENTREPRISE
par Dominique Helle et Guy Boussand,
256 pages, Réf. 0147

LOTUS 1-2-3 PROGRAMMATION DES MACRO-COMMANDES *par Goulven Habasque,*
144 pages, Réf. 0150 F

LOGISTÀT, ANALYSE STATISTIQUE DES DONNÉES
par Fredj Tekaia et Michele Bidel,
352 pages, Réf. 0115

ALGORITHMES *par P. Beaufils, et W. Luther,*
296 pages, Réf. 0149

POUR UN CATALOGUE COMPLET DE NOS PUBLICATIONS

FRANCE
6-8, Impasse du Curé
75881 PARIS CEDEX 18
Tél. : (1) 42.03.95.95
Télex : 211801

U.S.A.
2344 Sixth Street
Berkeley, CA 94710
Tel. : (415) 848.8233
Telex : 336311

ALLEMAGNE
Vogelsanger. WEG 111
4000 Düsseldorf 30
Postfach N° 30.09.61
Tel. : (0211) 61 80 2-0
Telex : 08588163

Paris • Berkeley • Düsseldorf

Achevé d'imprimer le 17 février 1986 sur les presses de l'Imprimerie «La Source d'Or»
63200 Marsat - Dépôt légal : 1er trimestre 1986 - Imprimeur n° 1975

— Tous les aspects essentiels de la programmation expliqués de façon logique dans un langage simple et clair.
— Une lecture facile : un texte fait aussi bien pour ceux qui n'ont jamais programmé, que pour les programmeurs plus expérimentés.
— De nombreux exercices pour contrôler vos connaissances et mettre en pratique les concepts présentés.

Ce qu'en dit la Presse Internationale...

— «Un livre excellent. Hautement recommandé» PET GAZETTE
— «Le style est clair et direct, et le contenu bien organisé» EDN
— «Tous les concepts de base sont présentés et expliqués. Le livre présente continuellement des détails en profondeur... Un livre pour un très large auditoire — du débutant au lecteur avancé» ELEKTRONIK
— «Les aspects essentiels de programmation et interface du 6502» 01 INFORMATIQUE

L'AUTEUR

a enseigné la programmation et les microprocesseurs à plus de 3 000 personnes dans le monde entier. Il est ingénieur de l'Ecole Centrale et docteur de l'Université de Berkeley où il a développé un interpréteur APL microprogrammé. Il a travaillé sur les applications industrielles des microprocesseurs à «Silicon Valley» dès leur apparition. Ce livre, comme les autres de cette série, est le résultat de son expérience professionnelle et pédagogique : il a publié plusieurs «best-sellers» sur les microprocesseurs, qui sont traduits en 10 langues.

0031 0286 128 F
9 782736 100315

SYBEX

RODNAY ZAKS

PROGRAMMATION DU 6502